ffice of cademic omputing

CONVERTING NSW FROM MVT TO MVS

NSW Semi-Annual Technical Report

October 1, 1980 - March 30, 1981

UCLA TR-29

Neil Ludlam
Denis De La Roca

niversity of alifornia, os ngeles

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: OAC/TR29	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER: AD-A186686	
4. TITLE *(and Subtitle)*: Converting NSW From MVT to MVS National Software Works	5. TYPE OF REPORT & PERIOD COVERED: Semi-Annual Tech. Report 10/1/80 - 3/80/81
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): N. Ludlam - D. DeLa Roca	8. CONTRACT OR GRANT NUMBER(s): MDA 903-80-C-0231
9. PERFORMING ORGANIZATION NAME AND ADDRESS: University of California, OFfice of Academic Computing, Los Angeles, CA 90024	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: Prog. Elemt: 62708E, Program Code: 0T10, ARPA Order No. 2543/12
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, VA 22209, Attn: Program Mgt. Office	12. REPORT DATE: 6/1/81
	13. NUMBER OF PAGES: 120
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

Distribution Unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

National Software Works, ARPANET, virtual remote batch terminals, virtual line terminal, MSG, PCP, PL/MSG, NSW architectural changes, NSW packages, file package, foreman, batch job package, virtual memory, interprocess communication, interactive debugging, process creation, process commulator, program logic.

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This report covers technical developments at UCLA-OAC relating to the National Software Works (NSW), during the period October 1, 1980, through March 30, 1981, and is specifically concerned with the conversion of existing software from the IBM System/360 MVT environment to the IBM System/370 MVS enviornment.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102 LF 014 6601

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

CONVERTING NSW FROM MVT TO MVS

by
Neil Ludlam

June 1, 1981

Document TR-29

UCLA Office of Academic Computing
5628 Math Sciences
University of California C0012
Los Angeles, California 90024

Except as noted herein, this work was sponsored by the Advanced Research Projects Agency of the Department of Defense, Under ARPA Order no. 2543, Contract Number MDA 903-80-C-0231:

ARPANET COMPUTER SERVICES IN SUPPORT OF THE NATIONAL SOFTWARE WORKS

April 9, 1980 - September 30, 1981

William B. Kehl, Principal Investigator
(213) 825-7511

SEMI-ANNUAL TECHNICAL REPORT
for period of
October 1, 1980 - March 30, 1981

REPORT SUMMARY

This report covers technical development at UCLA relating to the National Software Works (NSW) during the last quarter of 1980 and the first quarter of 1981.

The primary goal of the NSW project at UCLA is to make the IBM System/360 and System/370 family of computing systems "tool-bearing hosts" within the NSW. Previous phases of the project have made the UCLA implementation of the (now obsolete) operating system OS/MVT an NSW tool-bearing host. This phase of the project is to make the current IBM operating system, OS/MVS, a tool-bearing host.

This report is primarily concerned with the conversion of existing software from the MVT environment to the MVS environment.

The subsequent sections correspond to documents stored in the NSW documentation repository maintained by the NSW Operations Contractor, so each section has been made self-contained. For example, each section has its own reference summary and appendices.

CONTENTS

REPORT SUMMARY ii

PART I -- NSW Processes under MVS -- Basic plans

Section page

1. PERSPECTIVE 2

2. THE NEW ENVIRONMENT 4

Virtual Memory -- design opportunities 4
Inter-Address-Space Communications 4
Batch Job Scheduling 5
Interactive Job Scheduling 6
THREADER Commutator 6
NSW Architectural Changes 7
Relaxation of the TGET WAIT Problem 7

3. CHANGES TO NSW PACKAGES 8

PL/MSG 8
PL/PCP 9
MSG Direct Connections 9
The File Package 9
The Foreman 10
The Batch Job Package 11
Overall Design Strategy 12

REFERENCES 16

PART II -- THREADER -- A Commutator for PL/I Coroutines

Section page

1. OVERVIEW OF THREADER 19

Applications of the THREADER Package 20
How THREADER Relates to the Problem Program 22
Inter-thread Communication 24

2. THREADER SUBROUTINE CALLS 26

THSTART -- Starting a Subthread 26
THWAIT/THWAITL -- Letting Other Threads Run 28
THPOST -- Marking Events Complete 30

3. OTHER FACILITIES 31

MYTHB -- The Current Thread Handle 31
"PROCESS" -- PL/MSG Support 33
Managing ECB's in PL/I 34
ECBADDR -- Using PL/I Events 35

REFERENCES 36

PART III -- The UCLA Virtual Remote Batch Terminal Support Package

Section page

1. OVERVIEW OF THE VRBT 38

2. VRBT ADMINISTRATION 40

3. VRBT CHARACTERISTICS 41

4. THE VRTB PACKAGE ENVIRONMENT 44

5. VRBT RECORD FORMATS 46

6. VRBT DATA FLOW 47

Sending Data Sets 48
Sending Messages 51
Receiving Data Sets 51

7. SPECIAL SCHEDULING MODE 55

8. VRBT SUBROUTINE CALLS 57

VRBTOPN -- Materializing the VRBT 58
VRBTCLS -- Destroying a VRBT 60
VRBTSHT -- Requesting Shutdown 61
VRBTPUT -- Sending Data to JES2 62
VRBTGET -- Receiving Data from JES2 63
VRBTACK -- Providing Error Control 65

9. THE VRBT LOG 66

10. THE VRBT DEMONSTRATION PROGRAM 67

11. APPENDIX A -- PL/I DECLARATIONS 69

12. APPENDIX B -- ASSEMBLER DECLARATIONS 70

13. APPENDIX C -- JES2 INITIALIZATON PARAMETERS 75

14. APPENDIX D -- VTAM DECLARATIONS 76

15. APPENDIX E -- ALLOCATING VTAM ID'S 77

Calling Sequences 77
Allocating an APPLID 77
Freeing an APPLID 78
APPLALOC and the OPERATING SYSTEM 78
Allocation Technique 78
Defining the APPLID pools 78

REFERENCES 80

PART IV -- The UCLA Virtual Line Terminal Support Package

Section page

1. OVERVIEW OF THE VLT 82

Conversational Partners 83
Connection Protocol Type 1 83
Connection Protocol Type 2 83
The VLT Interface 84
Error Handling 85
Tasking Constraints 85
Ensuring Wait-free Operation 86
The VLT Package Environment 87
VLT Record Formats 87
Keyboard Management 88

2. VLT SUBROUTINE CALLS 89

VLTOPEN -- Materializing the VLT 89
VLTCLOS -- Destroying the VLT 92
VLTPUT -- Sending Data Through the Virtual Keyboard 93
VLTGET -- Receiving Data for the Virtual Printer 94
VLTATTN -- Pressing the Virtual Break Key 95
VLTCONT -- Sharing the Task With the VLT 96
VLTPRG -- Purging Pending Events 97
VLTKNT -- Counting Pending Events 98

3. THE VLT LOG 99

4. THE VLT DEMONSTRATION PROGRAM 101

5. APPENDIX A -- PL/I DECLARATIONS 102

6. APPENDIX B -- ASSEMBLER DECLARATIONS 103

7. APPENDIX C -- VTAM DECLARATIONS 108

REFERENCES 109

PART V -- FAKEMSG -- Debugging MSG Programs Without EXCHANGE

Section page

1. OVERVIEW OF FAKEMSG 111

2. STRUCTURE OF FAKEMSG 112

3. SPECIFIC PL/MSG SERVICES 113

MSGMP -- Materialize an MSG Process 114
MSGSTOP -- Destroy an MSG Process 114
MSGSETP -- Declare Posting Mechanism 114
MSGAA -- Arm Process for Alarms 114
MSGJOUR -- Log MSG Events 114
MSGRGM -- Receive Generic Messages 115
MSGRSM -- Receive Specific Messages 115
MSGRA -- Receive Alarms 115
MSGSGM -- Send Generic Messages 115
MSGSSM -- Send Specific Messages 115
MSGSA -- Send Alarms 116
MSGRSND -- Rescind Pending Events 116
MSGRSNC -- Resynchronize Communication 116
MSGOC -- Open Direct Connections 116
MSGCC -- Close Direct Connections 116
MSGGET -- Receive Connection Data 116
MSGPUT -- Send Connection Data 117
MSGEOD -- Prepare for Connection Close 117

4. CONCLUSIONS 118

REFERENCES 119

PART I

NSW PROCESSES UNDER MVS -- BASIC PLANS

This section is separately available
as UCLA Document UCNSW-213.

Section 1

PERSPECTIVE

In September, 1980, the UCLA Office of Academic Computing converted its IBM 3033 computing complex from the IBM operating system known as MVT ("Multiprogramming with a Variable number of Tasks") to that known as MVS ("Multiple Virtual Spaces"). While MVS is generally considered to be upward compatible with MVT, this is true only for programs which are entirely lacking in "system-intrusive" features.

For systems like the UCLA NSW system, this conversion presented significant incompatibilities and significant opportunities, due to considerations like these:

1. This conversion represented a change from real-memory-oriented operation to virtual-memory-oriented operation. This change offerred irresistable opportunities to improve some of the NSW code. It also represented a change from a system in which code in one "job" was able to reference data in another job if it was able to gain certain system privileges, to one in which individual address spaces are not only separated by rigid protection boundaries, but are logically unreachable from each other. For systems like NSW, which is implemented at UCLA as multiple jobs with much intercommunication, this caused serious problems.

2. Many of the operating-system interfaces required to implement a system like NSW were not present in MVT, and were added by UCLA systems programmers as system extensions. The MVS system, designed a decade later, includes most of these interfaces, but not, of course, in a form compatible with the UCLA extensions. Therefore, large sections of code designed around such interfaces were made obsolete.

3. The MVS system is, in general, better able to accomodate subsystems of the type of the NSW system than was MVT. It is possible to implement such systems in a way that is more robust and more maintainable, using the extended features of the new operating system. However, upgrading the quality of existing systems requires at least some reprogramming.

4. The NSW system conversion lay on a critical path behind the conversion of the UCLA inter-job communication mechanism, the EXCHANGE (reference 2). Delivery of that component to the NSW project was delayed for almost a year, due to problems arising in the MVS systems area. During that time the UCLA NSW implementation was not operational, and a number of architectural changes accumulated as a part of normal NSW evolution. Implementation of these changes thus became associated with the system conversion by default.

The material that follows represents an analysis of the problem of conversion of the NSW system from MVT to MVS, a statement of alternative design changes, and a basic plan for the actual conversion.

Section 2

THE NEW ENVIRONMENT

This section expands on the most important changes that will influence the redesign of NSW programs for the MVS environment.

2.1 VIRTUAL MEMORY -- DESIGN OPPORTUNITIES

Code written in PL/I to run under MVT/TSO tends to be organized primarily to conserve space and/or facilitate overlay. In some cases, this goal has actually directed software architecture. A goal of an MVS conversion project should be to identify and correct places where storage constraints have resulted in poorly designed programs. Of course, virtual memory is not a panacea for storage problems, and such changes should be tempered by the goal of keeping working sets small.

2.2 INTER-ADDRESS-SPACE COMMUNICATIONS

Inter-process communications, which were easily implemented under MVT, are more complex under MVS. Where communication is across job boundaries, it will now cross MVS address-space boundaries. This requires a different technique, such as intermediate buffering in a common storage area.

IBM supplies a program product, ACF/VTAM (reference 5), that accomplishes this function using intermediate buffering. However, the program interface to ACF/VTAM is complex, and good program design practices suggest that, to implement a simple inter-process comminications protocol, it should be front-ended by a package with a simpler interface.

The Johns Hopkins Shared Variable Manager (SVM) is another product that accomplishes inter-process communications through intermediate buffering (reference 18, 19). Its interface is simpler, but it will still benefit from a small front-end interface that offers primitives organized in terms of the problem at hand. SVM is deficient in that the upper limits on the traffic that it will handle are unreasonably small for a general-purpose communications mechanism.

ACF/VTAM has been demonstrated to be insufficiently robust to be used as an inter-process communications vehicle at this time. However, it remains the product of choice for the long run, due to its integration with the operating system, automatic maintenance by IBM, and lack of the limitations of SVM. SVM is the immediate choice for a temporary package.

In converting from an implementation that used the UCLA EXHANGE package (reference 2) under MVT, it will be wise to reduce or eliminate program dependence on features that are not fundamental to the general problem of inter-process communication. In the case of EXCHANGE, such features include:

1. Event signalling by software interrupt (the MVS "IRB" -- see reference 7).

2. Stuctured communications paths (EXCHANGE offers multiple "channels" per "window").

3. Discontiguous data areas (EXCHANGE provides scatter-read and gather-write).

4. Inessential status inquiries (EXCHANGE supports certain status inquiries by either process about the other).

5. A specific mode of process addressing (EXCHANGE uses a set of four 8-character "job" and "tag" names).

2.3 BATCH JOB SCHEDULING

The scheduling, monitoring, and retrieving of batch jobs under MVT used interfaces to the operating system's job queue that were developed as local system extensions. It is true that IBM provided some retrofitted capabilities of this sort to support their TSO time-sharing system, but these retrofits were too little and too late to be of great use at UCLA.

Under MVS, however, general purpose interfaces exist, and should be used. UCLA has implemented a Virtual Remote Batch Terminal, or VRBT (reference 12), using the IBM ACF/VTAM program product (reference 5). This virtual terminal allows any problem program in the system to connect to the MVS operating system as if it were a remote batch terminal.

The VRBT can and should be used to interface with the MVS batch job scheduling system. However, its interface is unique, and existing programs cannot be expected to use it without major changes.

2.4 INTERACTIVE JOB SCHEDULING

The logging-on of a time-sharing "pseudo-user" under MVT used interfaces to the operating system's telecommunications system that were developed as local system extensions.

Under MVS, however, general purpose interfaces exist, and should be used. UCLA has implemented a Virtual Line-oriented Terminal, or VLT (reference 11), using the IBM ACF/VTAM product (reference 5). This virtual terminal allows any problem program in the system to log onto the TSO time-sharing system as if it were a teletypewriter.

The VLT can and should be used to create interactive jobs as slaves of existing processes. However, its interface is unique, and existing programs cannot be expected to use it without major changes.

Notice that the VLT is not intended as a vehicle for MSG communication with NSW processes. Such communication requires a transparent binary path, such as provided by the EXCHANGE.

2.5 THREADER COMMUTATOR

UCLA has implemented a simple commutator package named THREADER (reference 13). This package allows the execution of multiple independent threads of control in programs written in PL/I. Using THREADER instead of PL/I multi-tasking simplifies the coding process due to simpler handling of critical sections. It may also avoid some of the overhead of true multi-tasking. It is useful in several circunstances:

1. When multiple instances of the same program are to be executed concurrently.

2. When multiple unlike programs could be executed together to advantage, as due to sharing some significant resource.

3. When a program is best designed as a set of co-routines.

2.6 NSW ARCHITECTURAL CHANGES

The NSW Architectural Control Contractor has proposed changes (reference 15) that allow local processes to execute in cooperation with each other, and to use each others' services without the intervention of the NSW Works Manager (reference 17) processes. These changes will be facilitated by some basic reorganization of the UCLA NSW code. We will include these reorganizations in our MVS implementation. Specific changes are covered in subsequent sections.

2.7 RELAXATION OF THE TGET WAIT PROBLEM

Under MVT, an interactive job that was waiting for terminal input was unavoidably and irrevocably blocked. This fact had significant impact on the design of any program that was to perform terminal interaction and other work concurrently. Under MVS, this restriction has been largely eliminated. This will mean that desirable program designs that had to be shelved under MVT can now be implemented, particularly in the area of the NSW Foreman.

Section 3

CHANGES TO NSW PACKAGES

NSW functional specifications allow the implementor great lattitude in choosing basic system structures. The UCLA implementation is quite modular, and can undergo basic structural reorganizations without serious effects on the bottom-level code.

This section ennumerates specific programming packages of the UCLA NSW implementation, and applies the aspects of the new environment to each. The most attractive options for conversion are presented.

In a final subsection, the options for the overall structure of the entire NSW system are examined.

3.1 PL/MSG

The PL/MSG package (reference 14) is the UCLA implementation of the process-to-MSG communications interface. All communication between an NSW process at UCLA and another NSW process passes through this package.

Under MVT, PL/MSG used EXCHANGE (reference 2) as its inter-process communication mechanism. Because WAITs are not permitted in PL/MSG code, and because that code must perform processing both before and after communication with EXCHANGE, not all such processing can occur during the term of a CALL to PL/MSG. The implementation uses software interrupt signalling (IRB signalling, see reference 7) as its mechanism for regaining control in order to perform post-EXCHANGE processing. It also makes use of the discontiguous data area features of EXCHANGE, although those features were used as a matter of convenience, not necessity.

Under MVS, programs using PL/MSG will be required to operate under THREADER. PL/MSG processing can be done on the thread of the caller, or on a thread dedicated to PL/MSG post-event processing. We will eliminate dependence on software interrupts and discontiguous data areas.

3.2 PL/PCP

PL/PCP (reference 10) is a subroutine package that defines and enforces a Procedure-Call protocol upon the basic PL/MSG primitives. Like PL/MSG, PL/PCP must perform processing at times other than those corresponding to CALL's upon its primitive routines. Since PL/PCP is written in PL/I, and since software interrupts are not generally available to PL/I programs, we resorted to defining extra calls to the PL/PCP package, and extra responsibilities upon the caller. The PL/PCP interface is thus cumbersome and unaesthetic.

Under MVS, programs using PL/PCP will be required to operate under THREADER. PL/PCP processing can be done on the thread of the caller, or on a thread dedicated to PL/PCP post-event processing. We will eliminate the unaesthetic parts of the PL/PCP interface.

3.3 MSG DIRECT CONNECTIONS

Under MVT, MSG direct connections (reference 16) were implemented as EXCHANGE windows. Since they were used by PL/I programs, and since EXCHANGE was a system service not callable in PL/I, an interface package, PLOXI (reference 1), was used. PLOXI was a completely general-purpose assembler-language interface between EXCHANGE and PL/I programs.

Under MVS, the inter-process communication mechanism will not be a system service, but a called routine. It will be called either directly from PL/I code or through a minimal assembler interface specific to the call. Since PLOXI was quite specific to the EXCHANGE, it will not be converted to MVS.

3.4 THE FILE PACKAGE

FP/360 (reference 3) was the MVT implementation of the NSW File Package process. Its lowest level is logically structured as a set of of co-routines. Previously, these coroutines were dynamically selected and bound together. Each such selectable routine was contorted into the form necessary to make it interface directly with the others. Under MVS, we will execute each such co-routine as a thread. There will be no need to invert the algorithms of the routines, so their logic can be simplified gradually as a part of routine module evolution. The resulting difference in the order of decision making will eliminate the need for dynamic selection and binding of the modules, simplifying the code and facilitating future maintenance. Because the FP/360 code is structured and modularized, these changes will not be massive.

There is a need to execute multiple File Packages concurrently. Previously, we have done this by logging on multiple TSO "pseudo-users" and invoking one instance of the File Package for each. We will now use THREADER to execute multiple instances of the same File Package code under one TSO logon. This will require only insignificant changes to the File Package code itself.

In summary, the File Package will become a single job, which will materialize as many MSG processes as are needed, or as are allowed by an initialization parameter. Since idle MSG processes will not represent significant allocated resources under this model, we can increase the number of File Package instances held at the ready considerably. The File Package job can be either a batch job or a TSO session.

3.5 THE FOREMAN

FM/360 (reference 9) was the MVT implementation of the NSW interactive Foreman process. It has always been a subset implementation due to the restrictions of the MVT environment. Under MVS, it can grow into a full implementation, although none of this growth need lie on the critical path to bringing NSW back up under MVS.

There is a need to support a "local name dictionary" (LND) to preserve the status of all interrupted tools across system or Foreman crashes. The LND can be shared by all active Foreman Processes, or there can be one for each Foreman instance. In either case, on system restart, a Foreman instance must report the status of all interrupted tools to the Works Manager. These needs could be met by operating multiple concurrent Foreman instances in a single address space, as dynamically created threads under THREADER, with a single LND-manager thread shared by all instances. Such a super-Foreman would be initiated at system restart, and at that time the LND-manager could perform the necessary Works-Manager communication.

There is a need to protect each Foreman from the tool which it supervises, since these tools are frequently undebugged. This need could be met by operating the Foreman and its tool in separate address spaces, communicating across the "thin-wire" communications path provided by an instance of the VLT. This design becomes more feasible when combined with the notion of housing all Foreman instances in a common job, as explained above.

There is a need for a Foreman process to select the workspace under which it will run its tool after the Foreman itself is already executing. This implies that the Foreman process is not itself "under" that workspace. Since we have always implemented NSW workspaces as TSO logon directories corresponding to TSO userids, this need also suggests that the Foreman should select a userid, log that userid onto TSO, and then invoke its tool program under that TSO session. The pool of available TSO userids could be allocated by the same thread that manages the LND.

There is a need to execute File Package processes directly from the Foreman. This need can be met merely by sending a newly defined procedure call from the Foreman to the File Package, or by merely executing the File Package code directly within the Foreman address space, using THREADER.

In summary, the Foreman will become a single job, which will materialize as many MSG processes as are needed, or as are allowed by an initialization parameter. Since idle MSG processes will not represent significant allocated resources in this model, we can increase the number of Foreman instances held at the ready considerably. The Foreman job can be either a batch job or a TSO session.

3.6 THE BATCH JOB PACKAGE

BJP/360 (reference 8) was the MVT implementation of the NSW Batch Job Package. The only part of its code that can be transferred to the MVS implementation is that concerned with PL/PCP transaction processing. The rest is concerned with interfaces to the MVT operating system which no longer exist.

BJP/370, the MVS implementation, will be designed around THREADER, the VRBT, and the UCLA Encapsulator Command Interpretor, or ECI (reference 4). The main thread of control will consist of the existing PL/PCP interface, a local job table manager, and code to create a VRBT-manager thread.

The VRBT-manager thread will create a VRBT and connect it to the MVS Job Entry Subsystem, JES-2. It will then create and coordinate threads to manage three virtual terminal devices: the card reader, the printer, and the operator's console.

The card-reader-manager thread will perform job submission. It will read SYSIN data sets that have been created by the Works Manager Operator (WMO) process (reference 8), pass them through the ECI, and pass the resulting jobstream into the card reader of the VRBT. Through the Tool Descriptor in the NSW Works Manager data base, the WMO will be configured to send SYSIN data in the form of ECI statements setting the various options that the user has entered. Through the tool-dependent ECI programs stored at UCLA, the ECI will be parameterized to expand these statements into MVS Job Control Language (JCL) statements (reference 6).

The printer-manager thread will accept job outputs from JES-2, spool them into SYSOUT data sets in the appropriate tool workspace, and generate notifications when the data is ready.

The console-manager thread will query or instruct JES-2 as needed by any other thread, including:

1. Query the status of a job, in response to a QUERY from the WMO.

2. Query the local name assigned the job in the card reader.

3. Query the local name of the job now on the printer.

4. Cancel a job.

5. Cancel a SYSOUT stream.

All these threads will request each others' services through a simple set of work-element queues.

At UCLA, there will always be only one BJP task per NSW system. It can be either a batch job or a TSO session. It will be initialized at system restart time. It will maintain a table of local batch jobs which is backed up on disk, and which is used to re-establish synchronism with WMO after system restart. It will use the ECI. In other words, it shares a great many attributes with the multiple-Foreman design mentioned earlier. There may be good reasons to combine the BJP and the Foreman into one job; however, we will not plan to do this at first.

3.7 OVERALL DESIGN STRATEGY

Combining all the notions developed in the previous sections, we see a design in which each supported generic process class is represented by a single job, either a batch job or a TSO session. Except for the BJP, which does not require multiple instances, the code in each job forks into as many identical MSG processes as specified by an initialization parameter, using THREADER.

Using this design, the process-spawning mechanisms of MSG Central (reference 16) will go almost unused. The main NSW jobs will be created by startup commands, rather than in response to a generic message.

There is no inherent reason why these three jobs cannot be combined into a single job. There are two areas of concern with that design. First, THREADER does not now support fault isolation to one thread, so that a program interrupt in one thread will bring down all. However, the additional THREADER code to support this will have to be written eventually. There is also the possibility that so many activities in one address space may result in a working set so large as to reduce NSW responsiveness. We will need experimentation to determine this.

The inherent flexibility of the NSW specifications and of the THREADER system allow decisions like these to be made very late in the design process. We will initially plan to use three TSO jobs; however, we can combine jobs or go to batch almost at will.

Figure 1: Basic NSW structure under MVT.

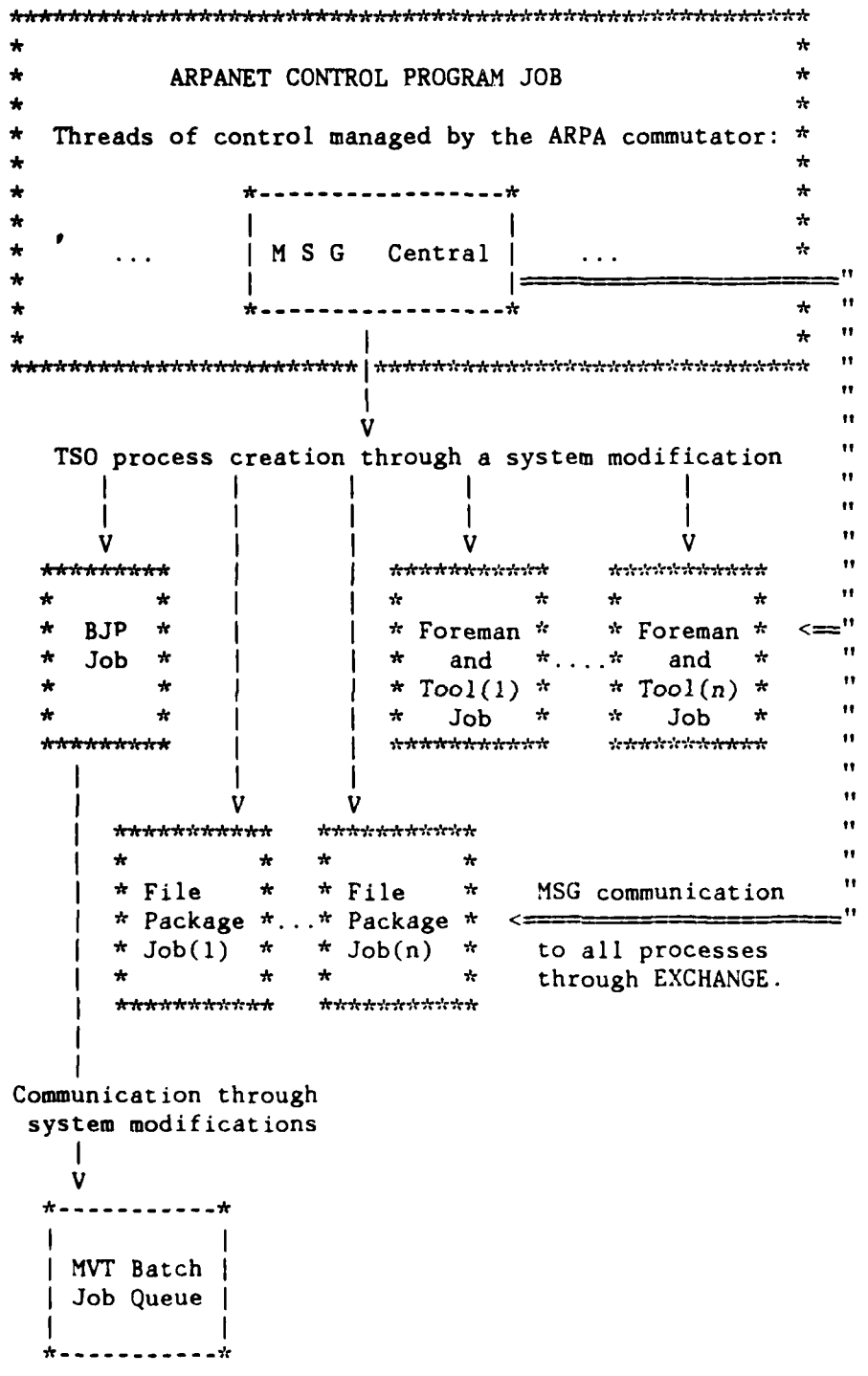


Figure 2: Multi-TSO-job NSW structure under MVS.

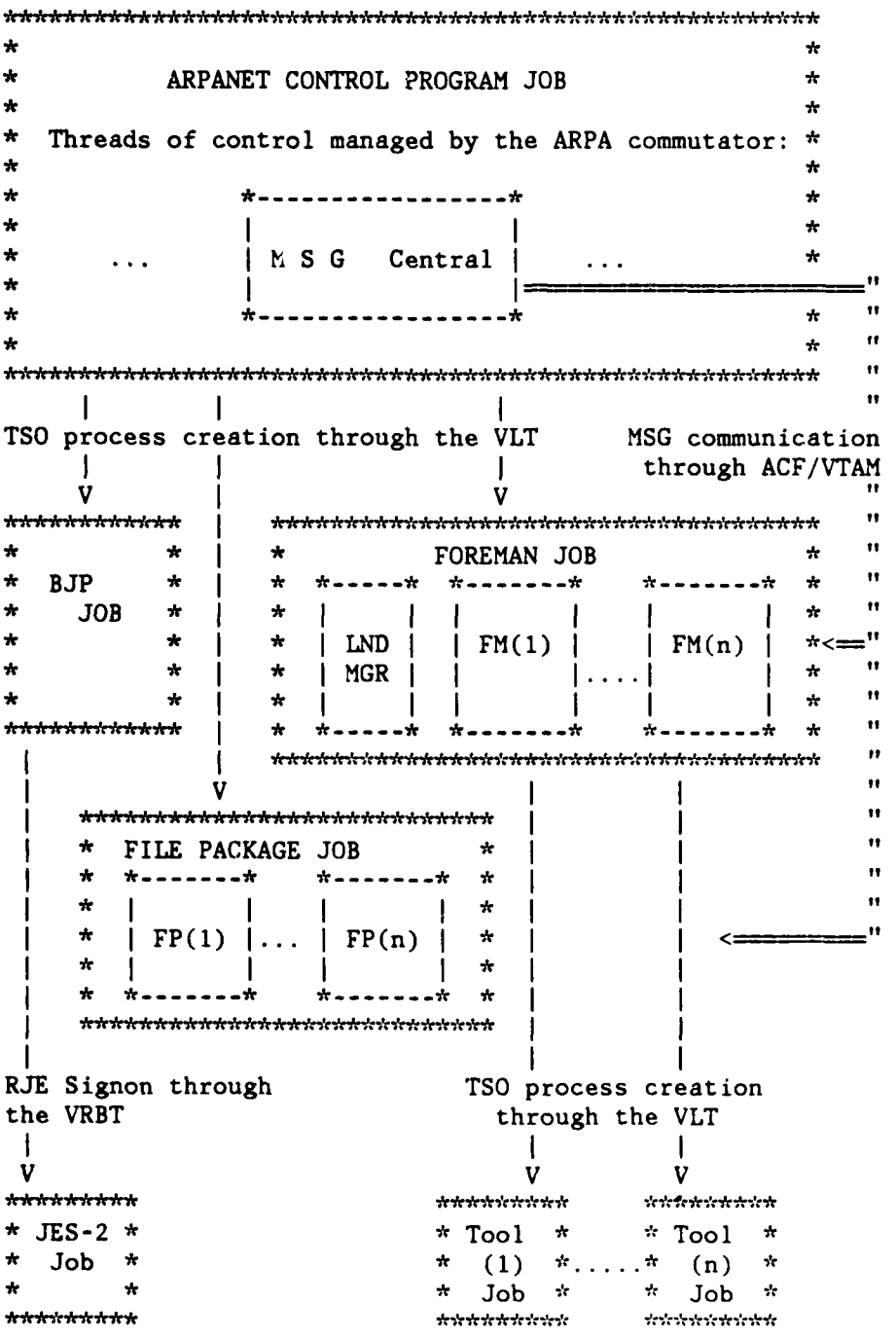
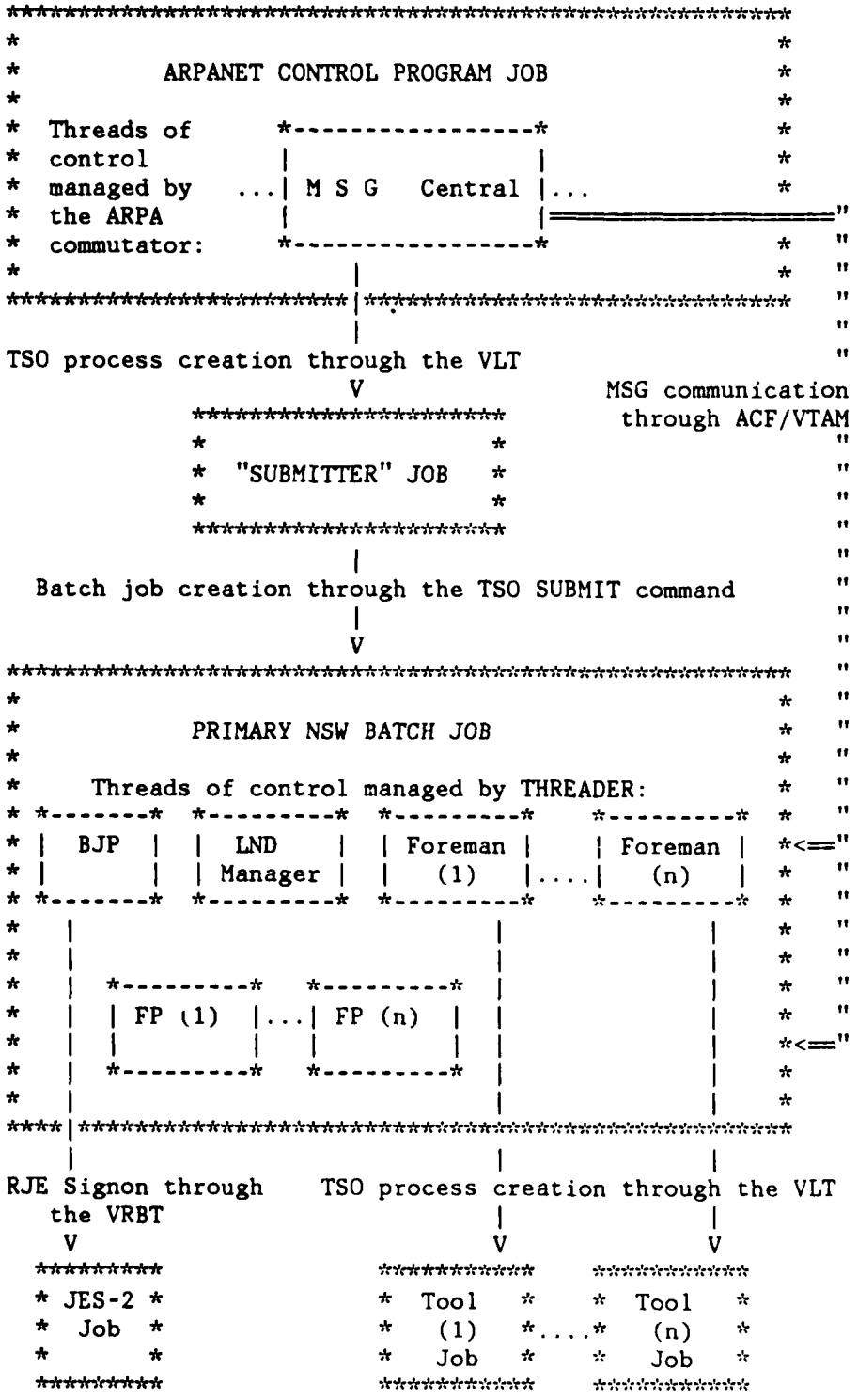


Figure 3: One-batch-job NSW structure under MVS.



REFERENCES

1 Braden, PLOXI -- A PL/I Interface to Exchange. Document UCNSW-407, Office of Academic Computing, UCLA, November 15, 1980.

2 Braden and Feigin, Programmer's Guide to the Exchange. Document TR-5, Office of Academic Computing, UCLA, March, 1972.

3 Braden and Ludlam, FP/360 - The NSW MVT File Package. Document UCNSW-204, Office of Academic Computing, UCLA, November 20, 1980.

4 De La Roca and Ludlam, The UCLA Encapsulator Command Interpretor System. Document UCNSW-206, Office of Academic Computing, UCLA, April 23, 1980.

5 IBM Corporation, Advanced Communication Function for VTAM: Programming. IBM Document SC27-0449, October, 1979.

6 IBM Corporation, OS/VS2 MVS JCL. IBM Document GC28-0692, May, 1979.

7 IBM Corporation, OS/VS2 System Programming Library: Supervisor. IBM Document GC28-0628, August, 1979.

8 Ludlam, BJP/360 - The NSW MVT Batch Job Package. Document UCNSW-207, Office of Academic Computing, UCLA, December 1, 1980.

9 Ludlam, FM/360 - The NSW MVT Foreman. Document UCNSW-205, Office of Academic Computing, UCLA, December 1, 1980.

10 Ludlam, PL/PCP - An NSW Procedure Call Protocol Package for PL/I. Document UCNSW-402, Office of Academic Computing, UCLA, November 15, 1979.

11 Ludlam, Programmers' Guide to the UCLA Virtual Line Terminal Support Package. Document TR-26, Office of Academic Computing, UCLA, February 1, 1981.

12 Ludlam, Programmers' Guide to the UCLA Virtual Remote Batch Terminal Support Package. Document TR-25, Office of Academic Computing, UCLA, February 1, 1981.

13 Ludlam, THREADER - A Commutator for PL/I Co-routines. Document UCNSW-409, Office of Academic Computing, UCLA, June 1, 1981.

14 Ludlam and Rivas, PL/MSG - An MSG Interface for PL/I. Document UCNSW-401, Office of Academic Computing, UCLA, November 15, 1980.

15 Massachusetts Computer Associates, NSW Exec Enhancements II. MCA Document P-3-019, November 26, 1980.

16 Rivas, Ludlam, and Braden, An Implementation of the MSG Interprocess Communication Protocol. Document TR-12, Office of Academic Computing, UCLA, May, 1977.

17 Schaffner and Sluizer, Works Manager Subsystem Specifications. Document CADD-7906-1117, Massachusetts Computer Associates, Inc., June 1, 1979.

18 ISI staff, SVM User's Guide. Document ISI10002, Interprocess Systems, Incorporated, Atlanta, Georgia, June, 1980.

19 IBM Corporation, VS APL for CMS: Writing Auxiliary Processors. IBM Document SH20-9068.

PART II

THREADER -- A COMMUTATOR FOR PL/I COROUTINES

This section is separately available
as UCLA Document UCNSW-409.

Section 1

OVERVIEW OF THREADER

THREADER is a subroutine package that enables multi-threaded execution of PL/I code, without the overhead and coding difficulties associated with multi-tasking. There are two main applications for this capability: co-routine design of application programs, and multiple simultaneous executions of a single application program. Combinations of these applications are also useful.

THREADER is designed to run co-operative, debugged programs. There is little error checking, and little attention given to fault isolation to a single thread, although such features could be added in future versions.

THREADER is not intended to be a full-fledged program monitor, nor to define a specific set of inter-thread communications protocols. It is a foundation on which such protocols may be built.

1.1 APPLICATIONS OF THE THREADER PACKAGE

Using THREADER, an application program can be designed as a set of cooperative co-routines, each performing a well defined function using a simple algorithm, and each communicating its needs and outputs to the others through rendesvous points that are essentially independent of the algorithmic structure. It is thus not necessary to invert an algorithm to, for instance, give it a single input point that corresponds to its procedural entry point. This capability has the potential of reducing debugging time, since an algorithm need only be tailored to the particularities of the data attributes of an application, not to the interactions of the various other algorithms with which it must communicate and coexist. Under these circumstances, it becomes more feasible to copy code from application to application with relatively minor changes.

THREADER differs from a multi-tasking environment in the degree of asyncronous operation. A given routine can be interrupted by any of its co-routines only at a point where it voluntarily relinquishes control through an external procedure call. Of course, all routines remain interruptible by truly asynchronous activities outside THREADER's realm of control.

This feature is useful in designing critical-section code such as that needed for inter-thread communications. In a multitasking environment, critical sections must be protected by some interlocking mechanism that prevents simultaneous accesses by concurrent threads. Such mechanisms are painful to program and irritating to use. Furthermore, it is difficult to demonstrate that every critical section is properly protected, since virtually identical program executions can produce drastically different patterns of interlock activity. Under THREADER, it is only necessary to ensure that critical sections are contained within a code segment that is not interrupted by an external procedure call.

As a further illustration of the difference between THREADER and multitasking, consider multiple concurrent execution of the same application program. Under multitasking, such a program must have all the properties generally grouped under the name "reentrant". Under THREADER, it need only have those properties collectively called "recursive".

THREADER is designed to run PL/I programs compiled by the IBM Optimizing Compiler (Reference 1, 2), and this documentation uses PL/I terminology; however, some use of other languages is permitted. In general, THREADER facilities may be called from any language capable of passing parameters in a compatible fashion. However, when a new thread of control is started by THREADER, a PL/I execution environment is created for it.

In general, multiple-thread design will use more virtual storage, because of the multiple incurrences of the initial space overhead of PL/I running-system dynamic areas such as the Task Communications Area (TCA). However, for applications where multiple independent programs

are combined under THREADER, there will be a space savings, since the same dynamic storage requirements will exist, but common copies of the modules from the extensive PL/I subroutine library will be used.

In general, we believe the processing overhead of using THREADER to be of similar magnitude to the savings realized by not coding inter-algorithm interfaces directly.

1.2 HOW THREADER RELATES TO THE PROBLEM PROGRAM

A THREADER application program consists of a main thread and any number of subthreads, which may or may not be further organized into subtree structures. A single application program is defined by the use of a single copy of the THREADER code. It is of necessity contained within one task, since the THREADER code is not itself reentrant. In other words, two applications cannot share one copy of THREADER.

THREADER is a subroutine package, not a main program. It creates subthreads, but it does not create the main thread from which it is called for the first time. So changing an existing program to use THREADER does not involve changing the way in which the program receives initial control.

THREADER is a substitute for PL/I mutli-tasking, where only one OS task is used, and the intertask communication facilities of PL/I are replaced by THREADER entry points. There are several important differences:

1. The multitasking versions of the PL/I library routines are not used.

2. Under multitasking, external procedures are usually declared REENTRANT. Under THREADER, they are usually declared RECURSIVE. There is a TASK option of the PL/I PROCEDURE statement; however, the current implementation does not require it for either multitasking or THREADER use.

3. Under multitasking, each PL/I task has associated with it a PL/I execution-time environment and an OS task. Under THREADER, there are multiple PL/I environments, but only one OS task. Therefore, language facilities associated with the PL/I environment exist multiply for threads. OS facilities associated with an OS task exist singly for all threads.

4. For example, CONTROLLED EXTERNAL storage is not shared among threads. PL/I files are addressed the same as CONTROLLED storage, and are not shared among threads. STATIC storage is shared among threads.

5. The STOP statement should terminate a thread, although at this writing this has not been tested. The effect of the EXIT statement is not now known. It should be avoided.

6. The OS facilities that are shared among threads include the system timer. For a PL/I program, the timer is directly available only through the DELAY statement. This statement will

delay all threads equally. In practice, OS interval timing services are often used from PL/I via Assembler subroutines. Such usage by more than one thread will cause unexpected behavior. Where this is a problem, the solution is to write a timer-management routine that executes as a thread. Such a routine is not provided as an integral part of threader, in keeping with the philosophy that THREADER is only a foundation for the building of more complex monitor systems.

7. Another system facility that is shared among threads is error control, as provided through the STAE and SPIE services. At this writing, THREADER forces all its subthreads to use the PL/I NOSTAE and NOSPIE options, and it depends upon its caller having done the same. This means that some PL/I error handling is disabled under THREADER. It is recognized that this is a serious omission, and we plan to re-enable these features in a later version.

8. The TIOT is shared, so if files are used multiply, they must usually have unique DDNAMEs (PL/I TITLE clauses). Files written simultaneously to the same DDNAME usually cause problems; however, if the DDNAME is allocated to a "shareable device," the data will be roughly interleaved. "Shareable devices" include the TSO terminal, real printers and punches, and JES2 SYSOUT files. The use of shared input files is probably not feasible.

In general, programs using THREADER should be declared RECURSIVE. The load module containing THREADER is neither reentrant nor read-only. THREADER assumes that it is the dominant control mechanism, and it uses static storage for its control data.

1.3 INTER-THREAD COMMUNICATION

Communication between threads is rooted in the list of parameters passed at thread-initiation time. Once two threads have established a common storage area they synchronize each other on a gross level through ECB's in the shared storage, and THREADER provides THWAIT and THPOST entries for this purpose.

The user of THREADER is assumed to be familiar with OS ECB management (reference 3), and with the essentials of PL/I subroutine communication (Reference 2).

THREADER ECB's are not PL/I EVENT variables, although a method is provided for interfacing with such variables if the PL/I program needs to do so. THREADER ECB's are just like OS ECB's, except that there are two types: external and internal.

1. An external ECB is an OS ECB. It is used to communicate between a thread and some external process. When THREADER finds all its threads blocked, its current set of external ECB's will be used in an OS WAIT SVC call. The handling of external ECB's is potentially completely asynchronous. If thread code is to post such an ECB, it must be done using a valid OS POST mechanism. THPOST is such a mechanism.

2. An internal ECB is just like an external one except that it is never used for any purpose except communication between two threads. Because such threads do not execute truly asynchronously, an internal ECB can be posted just by storing a value into it. Internal ECB's are never passed to OS in a WAIT SVC call. Therefore, it is permitted that more than one thread may be waiting on an internal ECB at a time. Such usage requires careful coordination of the threads themselves, but it is not considered unusual by THREADER.

While THREADER could be used to implement a system where all waits are strictly controlled, this is not a requirement of THREADER itself. Any thread is free to issue its own WAITs wherever it wishes. Of course, during such WAITs, no other thread can run.

Normal thread termination is by a normal return from the external procedure which was entered as the thread's main program.

The main program of the main thread, usually the "OPTIONS (MAIN)" proocedure of the load module, receives control BEFORE THREADER, and so is "started" and "terminated" differently from other threads.

THREADER keeps a tree of "Thread Control Blocks" (THB's) that define the current set of threads and their relationships. The address of such a block may be considered a "thread handle", and may be used to name the thread when communicating with THREADER. The handle of the current thread is always available in STATIC EXTERNAL POINTER variable MYTHB.

When any thread starts a subthread, the new handle will be returned in a parameter cell.

For many programs, thread handles are irrelevant, and can be ignored; however, use of such a handle in the THWAIT call can improve efficiency.

Parameters are passed to the main procedure of a subthread via the usual PL/I CALL mechanism. However, the call is to routine THSTART, which relays the parameters to the indicated routine. If THSTART is used in the same program to start more than one kind of subthread, it may be difficult to declare relayed parameters in any way except "*". When this is done, the caller will have to ensure that the arguments passed match the types expected, since automatic type conversion is not possible.

THSTART must be declared "OPTIONS (ASSEMBLER)". This means that normal PL/I data locator/descriptors are not used, and consequently, data that would normally require a locator/descriptor cannot be received directly by the called program. Such data include:

1. Character string data.
2. Bit string data.
3. Arrays of any kind of data.
4. Structures.

Of course, such data can always be passed indirectly by using POINTER parameters and passing arguments of the form "ADDR(...)", or by using based variables and ADDR in the called routine.

The parent thread will not continue execution until the subthread has been initiated and has returned control voluntarily to THREADER. When it does continue execution, the THSTART parameter list will usually be lost, since PL/I builds such lists in scratch storage. During this period, it is the responsibility of the subthread to capture all of its argument values that could possibly be passed in volatile storage.

Section 2

THREADER SUBROUTINE CALLS

2.1 THSTART -- STARTING A SUBTHREAD

THSTART is called by any thread to establish a subthread, to start it running, and to pass it parameters.

```
DECLARE THSTART ENTRY (CHAR(8), ENTRY,
                       FIXED BIN(31), POINTER,
                       FIXED BIN(31), *, *,...)
                  OPTIONS (ASSEMBLER, INTER);
CALL    THSTART (name, entrypoint, isasize,
              handle, termecb,
              parm1, parm2, ...);
```

Where:

1. "name" is an 8-character name to be assigned to the subthread.

2. "entrypoint" is an ENTRY constant or variable identifying a PL/I EXTERNAL procedure or entry, which need not be declared "OPTIONS (MAIN)." This will be the main procedure of the subthread. It is also possible to declare this parameter POINTER, when that facilitates the calling program's usage.

3. "isasize" is as described in the PL/I programmer's guide (reference 2). The absolute minimum here seems to be about 2200. Most programs will want much more.

4. "handle" is a POINTER variable which will receive the subthread handle.

5. "termecb" is a fullword to serve as a subthread termination notification. THSTART will clear the post bit of this word, and will then "post" it when the subthread has terminated. This might be as early as before control returns from the THSTART call. The post code will be the contents of register 15 on exit. When this ECB is posted, the subthread handle becomes invalid.

6. "parm1", "parm2", etc. stand for the parameters to be passed to the subthread entry point. There can be any number, including zero, and their types can be whatever the called routine will expect. See the restrictions on this list under "INTER-THREAD COMMUNICATION".

Parameters may be omitted from the right end of the list, when this is appropriate, but the isasize and everything to the left of it are required. The termination ecb and/or thread handle parameters may be designated as omitted even when there are specified parameters following them. This is done by passing a parameter ADDRESS (not VALUE) of ZERO (not NULL). This is not easily accomplished in PL/I, but it can be done by this artifice:

```
DECLARE ZERO       FIXED BINARY (31) STATIC INITIAL (0),
        ZEROPTR    POINTER BASED (ADDR (ZERO)),
        NULLHANDLE POINTER BASED (ZEROPTR),
        NULLECB    FIXEC BINARY (31) BASED (ZEROPTR);

CALL THSTART ( ... NULLHANDLE, NULLECB, ...);
```

When there is a termination ECB, the new thread will be a subthread of the calling thread. Note that posting of the termination ECB is always suppressed if the parent thread has already terminated. When there is not a termination ECB, the new thread will be a subthread of the main thread.

When there is no handle variable, the subthread handle is not returned to the caller, but there are no other side effects.

2.2 THWAIT/THWAITL -- LETTING OTHER THREADS RUN

THWAIT and THWAITL cause the calling thread to be blocked until some event occurs. In the meantime, other threads will execute. THWAITL differs from THWAIT only in its syntax. It is provided to support situations where the set of ECB's is too dynamic to fit the simple syntax of THWAIT.

The caller may recommend a thread to be given control if it wishes. If it does so, and if that thread is indeed ready, it will be the next thread to run. If no recommendation is made, or if the recommeded thread is not ready, THREADER will choose a thread to run, using a strictly round-robin technique.

When a ready thread is reccommended, thread-switching overhead is of the same order as subroutine calling. Otherwise, thread-switching overhead is unpredictable, and depends on THREADER load. Hence it is important to make a valid recommendation whenever possible.

The caller may specify any number (up to 64) of events to wait for. When any one is posted, the caller is again eligible to receive control. If any specified event is already posted, the caller is eligible to receive control again at once; however, this will happen only if there is no other ready thread to be run. It is legitimate to pass no ECBs. This is equivalent to passing a single pre-posted ECB.

It is up to the caller to clear all ECBs before they are associated with an event. THREADER cannot clear ECBs, since it is legitimate for them to be posted already.

The caller is also responsible for distinguishing between external and internal ECBs. THREADER has no other way of knowing this, and if an external ECB is mistakenly treated as internal, the entire system may lock up.

```
DECLARE THWAIT ENTRY (POINTER, FIXED BIN(15),
               FIXED BIN(31), FIXED BIN(31), ...)
               OPTIONS (ASSEMBLER, INTER);
CALL    THWAIT (recommendation, externalcount,
                ecb1, ecb2, ...);

DECLARE THWAITL ENTRY (POINTER, FIXED BIN(15),
               FIXED BIN(15), (*) POINTER(31))
               OPTIONS (ASSEMBLER, INTER);
CALL    THWAITL (recommendation, ecbcount,
                 externalcount, ecblist);
```

Where:

1. "recommendation" is the handle of a thread being recommended as the next one to receive control, or is NULL if no recommendation is being made. This parameter may not be ommitted.

2. "externalcount" is the number of external ECBs in the following list. These must be the first ones named. The remainder of those named will be treated as internal ECBs.

3. "ecb1", "ecb2", etc. are the ECBs representing the events being waited for. There may be as few as zero or as many as 64 of these. If there are zero, the "externalcount" can also be omitted.

4. "ecbcount" is the number of entries in "ecblist".

5. "ecblist" is a vector of "ecbcount" ecb addresses. The first "externalcount" of them are external, and the remainder are internal. There is no form of list terminator other than these separately-stated counts.

6. If no ECBs are being passed to THWAIT, then only the "recommendation" parameter need appear.

2.3 THPOST -- MARKING EVENTS COMPLETE

The THPOST routine is provided only as a programming convenience. Any valid interface to OS POST can be used to mark external events complete. Internal ECBs can also be POSTed; however, it is just as proper (and considerably faster) to mark them complete with an assignment statement. For efficiency, THPOST uses "fast post" (the CS instruction) when possible, and the OS POST SVC call otherwise.

```
DECLARE THPOST ENTRY (FIXED BIN(31),
                      FIXED BIN(31))
                OPTIONS (ASSEMBLER, INTER);
CALL THPOST (ecb, code);
```

Where:

1. "ecb" is the ECB to be posted.

2. "code" is the post code. Only the lower 30 bits are used, since the top two bits of an ECB are used for event status. This parameter may be omitted, in which case zero will be used.

Section 3

OTHER FACILITIES

3.1 MYTHB -- THE CURRENT THREAD HANDLE

The MYTHB datum is the only way a thread normally knows its own handle. It always contains the handle of the currently executing thread. You may use it, but you may not change it.

```
DECLARE MYTHB STATIC EXTERNAL POINTER;
```

In fact, MYTHB is the name of the THREADER control section, and it can be used to reference other data of interest by a declaration of the form:

```
DECLARE 1 MYTHB STATIC EXTERNAL,
          2 THBADDRESS  POINTER,
          2 PLI_OPTIONS,
            3 (REPORT, NOREPORT, SPIE, NOSPIE,
               STAE, NOSTAE, COUNT, NOCOUNT,
               FLOW, NOFLOW, FILL (22)) BIT(1),
          2 MODIFIERS,
            3 (DOPURGE, TESTING, PRINTNEW,
               FILL (29)) BIT(1);
```

The PLI_OPTIONS bits have the meanings of the corresponding options of the EXEC card (reference 2). Just one of each pair must be set to 1. They are applied when a new thread is started. You can set them at will; however, THREADER will force NOSPIE and NOSTAE when a new thread is started.

The MODIFIERS are for you to set. They have these meanings:

1. DOPURGE can be set to cause THREADER to terminate and purge all subthreads of a terminating thread. At this writing this feature is untested.

2. TESTING can be set to cause THREADER to TPUT the name and address of every newly created or destroyed THB, and the name of every THB that is dispatched.

3. PRINTNEW can be set to cause THREADER to TPUT the name and address of every newly created or destroyed THB.

As in any case where a STATIC EXTERNAL datum is declared in a less than complete form, declarations like these should not be allowed to supply the actual datum when the program is linkage-edited. To ensure this it is only necessary to include THREADER in the linkage-editor input before the program in which the declaration occurs.

3.2 "PROCESS" -- PL/MSG SUPPORT

THREADER is expected to be used with the PL/MSG subroutine package (reference 4). In its present form, that package uses a STATIC EXTERNAL POINTER datum named PROCESS to identify the current MSG process. THREADER includes a data ENTRY named PROCESS, which it will maintain across thread swaps. Thus PL/MSG need not be made reentrant to use THREADER.

Programs that do not use PL/MSG need not be concerned with THREADER's support for the package. It is transparent to such programs, and it does not add measurable overhead.

If PL/MSG is being used, it is necessary to ensure that the copy of PROCESS that is included in a load module is the one provided by THREADER, and not one from another object module. To ensure this, it is only necessary to include THREADER in the linkage-editor input before any programs in which PROCESS is declared.

3.3 MANAGING ECB'S IN PL/I

THREADER ECB's are not normal PL/I data, and managing them in PL/I can be a problem. The following techniques can be used:

```
DECLARE POSTED FIXED BIN (31) INIT ((2**30));
```

1. To clear an ECB:

```
ecb = 0;
```

2. To "post" an internal ECB:

```
ecb = POSTED;
```

3. To test an ECB for completion:

```
IF ecb >= POSTED
   THEN  action for posted event;
   ELSE  action for pending event;
```

4. To test a posted ECB for a zero postcode:

```
IF ecb = POSTED
   THEN  action for zero postcode;
   ELSE  action for non-zero postcode;
```

5. To extract the postcode from a posted ECB:

```
code = ecb - POSTED;
```

3.4 ECBADDR -- USING PL/I EVENTS

If a thread program must use a PL/I EVENT variable, it can use the (untested) ECBADDR function to extract the corresponding ECB address for passing to THREADER. For example:

```
DECLARE ECBADDR ENTRY (EVENT) RETURNS (POINTER),
        ECBWORD FIXED BIN(31) BASED;
CALL THWAIT (...,
        ECBADDR(EVENTNAME)->ECBWORD, ...);
```

ECBADDR is intended for use with ACTIVE I/O events. It must be emphasized that not all PL/I EVENT variables always contain the address of an ECB. It is up to the programmer to understand PL/I's use of ECB's, and to ensure that this routine is not misused. The ECBADDR makes no checks, and if it is used at an inappropriate time, it will return garbage.

REFERENCES

1 IBM Corporation, OS PL/I Checkout and Optimizing Compilers: Language Reference Manual. IBM document GC33-0009, October, 1976.

2 IBM Corporation, OS PL/I Optimizing Compiler: Programmer's Guide. IBM document SC33-0006, October, 1976.

3 IBM Corporation, OS/VS2 MVS Supervisor Services and Macro Instructions. IBM document GC28-0683, April, 1978.

4 Ludlam and Rivas, PL/MSG -An MSG Interface for PL/I. Document UCNSW-401, Office of Academic Computing, UCLA November 15, 1980.

PART III

THE UCLA VIRTUAL REMOTE BATCH TERMINAL SUPPORT PACKAGE

This section is separately available
as UCLA Document TR-24.

Unlike other sections of this report, this section does not describe work supported by the Department of Defense. This work was supported by the Office of Academic Computing of the University of California at Los Angeles. It is included here because it describes work that will be used extensively by the NSW project, and because it lays the foundation for other sections which do describe work supported by the Department of Defense.

Section 1

OVERVIEW OF THE VRBT

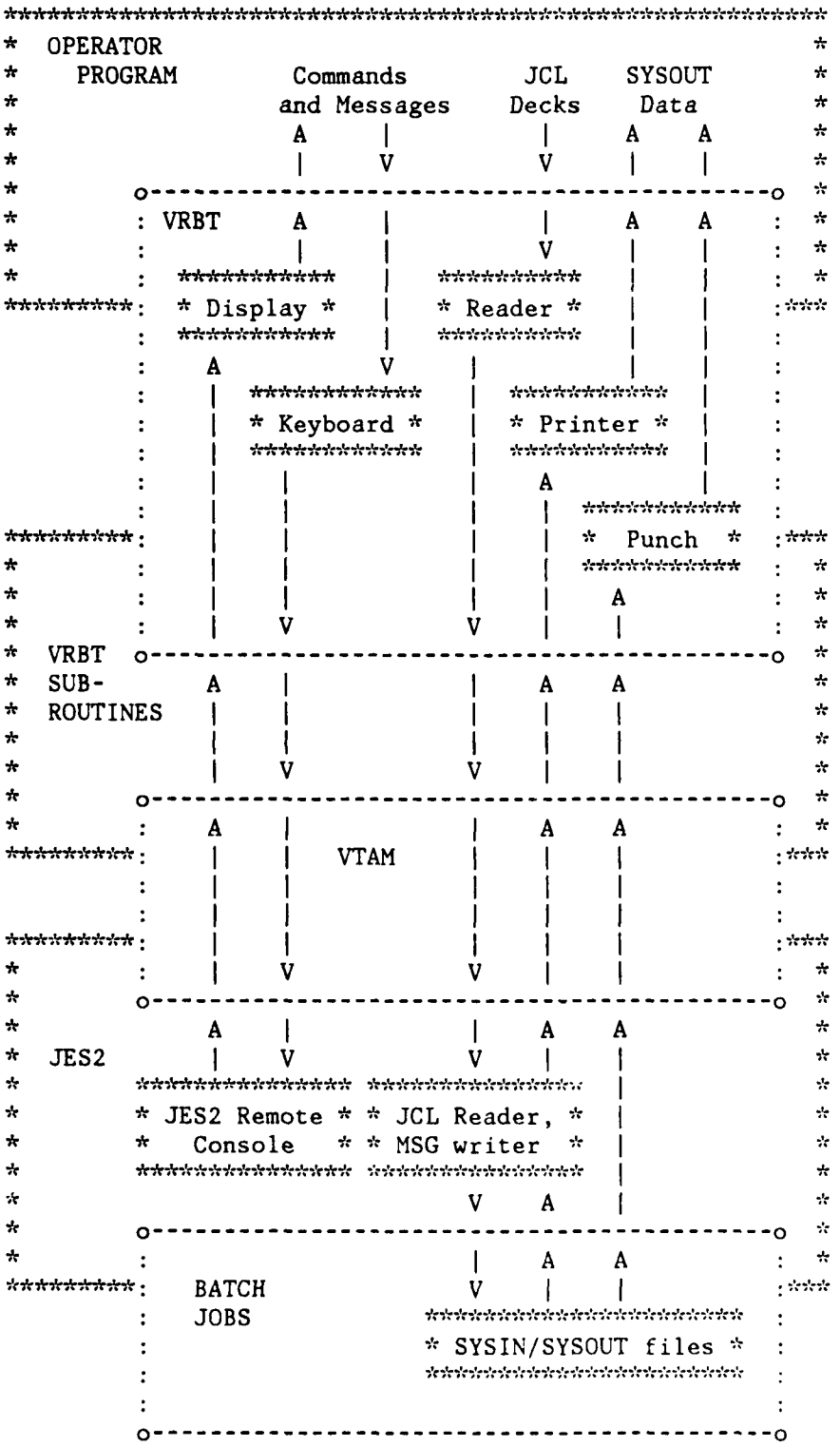
The UCLA Virtual Remote Batch Terminal (VRBT) package simulates remote batch terminals connected to JES2 (reference 5). It is a reentrant subroutine package that creates, manages, and destroys subtasks of its caller. Each such subtask is a VRBT.

The calling program is the operator of the VRBT. It can read cards through its card reader, print job output through its printer, and punch output decks through its punch. It can enter JES2 commands (reference 4) through its keyboard and monitor JES2 replies through its console display. These five virtual devices are refered to as "channels". In addition to managing these channels, a VRBT operator program can do anything it wishes. For instance, it may interface JES2 to unsupported RJE sources (reference 1), or it may support the submit/retrieve functions of a private workbench system.

The calling program uses VRBT entries to materialize and destroy an entire VRBT. Once the VRBT is successfully materialized, all other operations directed toward it are specific to a channel. The operator program is free to manage all five channels simultaneously, or he may limit data transfer to one stream at a time. Usually, though, he will want to monitor the console channels at all times.

A schematic illustration of the relationship of a VRBT and the VRBT support subroutines is shown in Figure 1. The purpose of the support subroutine package is to isolate the VRBT operator program from the peculiarities of JES2 data stream handling and VTAM transmission protocols (reference 2). A lot of things can go on while operating a VRBT, and the operating program should not have to deal with more complications than are necessarily involved in managing virtual devices.

Figure 1. Schematic of a VRBT.



Section 2

VRBT ADMINISTRATION

The materialization of a VRBT requires as parameters the Remote Terminal ID (or RMTID) by which the VRBT is known to JES2, and the password, if any, that JES2 associates with that ID. These values are specified by a systems programmer when he generates the JES2 configuration file. Their assignment to users is an administrative function, and will be handled differently by different installations. The VRBT user should consult his systems staff and/or user-services staff for a permanent and exclusive assignment of an RMTID and password.

The RMTID and password are those specified on a RMTnn parameter of the JES2 initialization data set. Likewise, the terminal ID is NOT the "luname" subparameter of the RMTnn parameter, or the string of the form "RMTnn", but rather the actual number "nn". JES2 uses the RMTID as the vehicle for partitioning all its job-management activities. The user of an RMTID effectively sees the operating system as though only those jobs submitted from and/or bound to his RMTID existed, and he has little control, on the VRBT level, of when output from these jobs is returned. For these reasons, it is advisable that an RMTID be permanently assigned to a user or to a small group of users. However, if the VRBT is to be managed by an operator program which itself performs job-management functions, then an RMTID, or a pool or RMTID's, should be permanently assigned to the class of RJE functions managed by that program.

VRBT materialization also requires the use of an ACBNAME known to VTAM. This name is chosen by the VRBT support subroutines by using the external entries APPLALOC and APPLFREE, specifying name pool 0. These entries are separately documented in an appendix. They include an assembled-in pool of ACBNAMEs, and maintenance of this name pool is an administrative function.

Section 3

VRBT CHARACTERISTICS

1. The VRBT has a fixed configuration. It has five channels, assigned as follows:

   Channel 1 is the console display device.
   Channel 2 is the console keyboard.
   Channel 3 is the card reader.
   Channel 4 is the printer.
   Channel 5 is the card punch.

   These values are defined by assembler macro VRBTEQUS, which is listed in Appendix B.

2. JES2 compression and compaction are never supported. Transparency is always supported.

3. From JES2's point of view, the display and keyboard are always active. The other channels can be activated and deactivated by JES2 commands entered through the console. The initial states are active.

4. The VRBT can transfer logical records of up to 255 bytes. In normal operation, records will be no longer than 80 bytes for card reader data, or 255 bytes for other media.

5. Data transfer is done on a data set basis. It is not possible to isolate errors to a lower level. All errors result in the aborting of the data set being transmitted at the time. For the printer, punch, and display streams, data set boundaries are specified by JES2. For the card reader stream, data set boundaries are specified by the operator program. For the keyboard, each message is considered a data set.

6. The materialization of a VRBT requires as parameters an RMTID and a password. The VRBT user must have these values assigned him by his installation. See the section entitled VRBT ADMINISTRATION for details.

7. The VRBT package provides routines for opening and closing an entire VRBT, and routines for getting data from and putting data to a specific channel. The get and put routines allow the operator program to perform multiple buffering if he wishes. The object of a put, or the result of a get, may be either a data record, an end-of-data-set signal, or an abort-data-set signal.

Only data is normally passed through the keyboard channel, since data set boundaries occur implicitly after each message.

8. The end-of-data-set or abort-data-set signals provide resynchronization of a channel stream. Thus errors may be isolated to a single data set. The keyboard channel should always be synchronized, so that errors can be isolated to a single message.

9. Therefore it may be wise, where practical, to send each input job as a separate data set. This ensures that, in the event of an error, the sender knows what jobs were successfully submitted. But in any case, the command stream can be used to determine this.

10. In order to prevent an output data set from being lost on the output end of the VRBT operator program and still deleted by JES2, an end-of-data-set signal on a JES2-to-VRBT data set requires a specific confirmation by the operator program. Pending this confirmation, the VRBT will not acknowledge receipt of the dataset to JES2, JES2 will not purge the data set, and no further data transfers will occur on the channel. The VRBTACK routine is the mechanism for indicating this acknowledgement. The operator program should call that routine in response to an end-of-data-set signal, but not until it has been determined whether the data set has been disposed of in a satisfactory way. Once the end-of-data-set has been positively acknowledged, the JES2 copy of the data will be purged.

    VRBTACK is required on those channels which transmit true output data sets, namely the printer and punch channels. It is not required on the console channel, and should not be used there.

11. There is one exception, called "shutdown", which applies to the VRBT as a whole. It is signalled by the VRBT to the operator program through an ECB dedicated to that purpose. Once this ECB is posted, the operator program should voluntarily close the VRBT as soon as all current data sets are completed. The posting of the ECB does not, in itself, affect normal operation of the VRBT; however, a total loss of communication with JES2 will appear as a shutdown signal posted simultaneously with abort-data-set conditions on all active channels.

    The values of the codes posted in the shutdown ECB are defined by assembler macro VRBTABCD, which is listed in an appendix.

12. Symmetrically, the operator program can call the VRBTSHT entry to request that JES2 initiate an orderly shutdown. Calling this entry has no effect on data sets already in progress. It may result in the posting of the shutdown ECB.

13. All VRBT subroutines return control to the caller at once, with completion of the requested operation signalled by the later posting of an ECB.

14. The low-order 30 bits of a posted ECB receive a status code. There is a general convention that zero reports normal completion, a positive value (usually 1) reports an end-of-data-set condition, and a negative number (usually -1) reports an error. The VRBT routines will always use fast post (the CS instruction) where possible.

15. The VRBT functions as a subtask of the caller of the "open" subroutine, which issues an ATTACH macro-instruction. Other VRBT routines can be called by that task or by any other task; however, the routines are serially reuseable, and are NOT protected from use by concurrent callers. Therefore, if the operator program uses multiple tasking, it should provide its own interlocks. The only communications between the VRBT subtask and other tasks are the entries and ECB's described in this document.

16. Except for the VRBTOPN routine, the subroutines cause very simple task synchronization, and in normal scheduling mode, may issue WAIT and POST. WAIT is for a CPU-bound event in the VRBT subtask, and should not be able to delay the calling task. This mode of operation should be satisfactory in all environments except where WAIT and POST are prohibited for reasons other than delay.

17. However, for operator programs whose environment absolutely prohibits the use of WAIT and POST, a special scheduling mode is defined. In this mode, the ECB's requiring WAIT and POST are passed back to the calling program, where they can be treated in whatever way is defined in that environment. These ECB's never expect or receive a status code, so the operator program need not indicate nor check for exceptions. Note that the special scheduling ECB's are not in any way related to the ECB's that are used to signal the completion of a requested event. Those latter ECB's are always under the control of the operator program.

18. All VRBT entries are PL/I (optimizing compiler, reference 8) compatible through the "OPTIONS (ASSEMBLER)" facility. Canned declarations are provided for both the PL/I and the Assembler caller. Because the reasons for using special scheduling mode are quite incompatible with the use of PL/I, the canned PL/I declarations do not include data used only in that mode. The PL/I declarations treat ECB's as "FIXED BIN (31)" data.

Section 4

THE VRTB PACKAGE ENVIRONMENT

The VRBT support subroutines consist of five control sections:

1. VRBTSUBS contains the user-visible entry points. It is about 1200 bytes long.

2. VRBTTASK contains the mainline of the subtask code. It is about 3300 bytes long.

3. VRBTSTRS contains internal routines for the subtask. It is about 9000 bytes long.

4. VRBTTABS contains code and tables to support the formatting of VTAM RPL's for the VRBTLOG function. It can be placed in such a way by the linkage edit that it will remain paged out when logging is not enabled. If you can guarantee that logging will never be enabled, it can be deleted from the load module. It is just under 4K bytes long.

5. VRBTDUMP contains code to support debugging under the TSO TEST debugger. It is not useful in any other context. It can be deleted from the load module without causing an unresolved external reference. It is about 256 bytes long.

In addition to the static storage occupied by these control sections, each materialization of a VRBT will acquire approximately 16K bytes of working storage from subpool 111. This storage is released when VRBTCLS is posted complete.

In the process of materializing a VRBT, IDENTIFY and ATTACH are used to initiate the subtask. The entry that is the object of the IDENTIFY is VRBTMAIN, which is an entry of VRBTTASK. It is necessary that this entry point be available to ATTACH through one of three mechanisms:

1. If the load module containing VRBTMAIN is a legal load module according to the rules of IDENTIFY, then an IDENTIFY will be issued, and the entry point will be available. No check is made to see if the IDENTIFY fails, so if VRBTMAIN is already on the load list, no harm is done.

2. If the load module is still available from a library that is known to ATTACH (system linkage library, JOBLIB, STEPLIB, or task library), and if VRBTMAIN is an alias, then the entry point will be available. The directory scan will be done under the subtask, and will not cause the caller of VRBTOPN to wait.

3. Otherwise, the operator program should perform whatever operations are indicated in his environment to get VRBTMAIN on the load list for his task.

Allocation and freeing of the ACBNAME used to materialize a VRBT is accomplished through LINK to external entries APPLALOC and APPLFREE. Because these entries constitute a system-wide pool of ACBNAMES that is subject to update, there should only be one copy of them in the system; therefore, they should not be linkage-edited into any load module. It is the calling program's responsibility to ensure that they are available to the LINKs. Note that the LINKs are executed under the subtask, and cannot cause the operator program to wait.

Section 5

VRBT RECORD FORMATS

All data records handled by the VRBT use the same basic representation. A record is a simple character string containing only data characters. There are no compression or transparency control characters, and no device control characters except for ANSI control characters in column 1. The record is thus completely described by a pointer to its first character (PL/I users note: NOT to a count field) and a count of the number of characters.

Records are always pruned of all trailing blanks, either by JES2 or by the VRBT subroutines. Blank pruning is without regard to whether the data is textual or binary, so if the operator program extends records output by JES2, even a "binary card" from the punch stream should be extended to full length by EBCDIC blanks (X'40'), not zeros.

Certain characteristics vary by channel:

1. The printer and punch streams can receive records as short as one character, or as long as 255 characters. The first character is always an ANSI carriage (or stacker) control character. It is expected that most job output will be limited to 133 characters for printer data and 81 for punch data, but the VRBT doesn't enforce this. The operator program can truncate or blank-pad as it sees fit. Zero-length records are represented as a single carriage-control character (one blank).

2. The card reader stream is going to be interpreted by JES2 as 80-byte card images without ANSI control codes. Therefore the VRBT truncates any longer record to 80 bytes before pruning trailing blanks. Truncation is not considered an error condition.

3. The keyboard and display streams consist of records as short as one character or as long as 255 characters. There are no ANSI control codes. Zero-length records are simply deleted from the display stream; therefore, the operator program should not assume any vertical formatting of any kind in this stream.

Section 6

VRBT DATA FLOW

The JES2 VRBT consists of five channels. The keyboard channel is message oriented, in order to simulate an operator's console. Keyboard data transfer is not posted complete until it is accepted by JES2. If a transmission error occurs, the operator program is notified, but previous and subsequent transmissions are not directly affected.

The other four channels are data-set oriented. A display "data set" can be as short as one record, especially since the VRBT deletes empty records from that stream. While data flow is scheduled on a logical-record basis, error control is on a data-set basis. If a transmission error occurs, the operator program is notified, but the notification should not be interpreted as related to any particular data record, but rather to the data set as a whole. There is no recovery defined for such an error.

The virtual console is handled as though it were not locking. If a locked keyboard is needed, the operator program should lock the keyboard when VRBTPUT is called to send keyboard data, and unlock it when an EOD signal is received on the display channel.

6.1 SENDING DATA SETS

Once the VRBT is open, the operator program schedules JES2 input data sets (card reader data) by calling VRBTPUT, passing a VRBT queue element (VQE) which contains an ECB. The program can schedule as much input on a channel as it has VQE's to dedicate, with the effect of arbitrarily deep buffering. If multiple buffering is used, the operator program must check the completion status of all buffers in the order that they were queued. Data is always transmitted in that order, and the posting of error conditions that affect the state of the channel is queue-order dependent. The operator program should provide its own ordering mechanism, perhaps through an extended VQE format. It should never attempt to use or interpret the internal chaining fields of the VQE itself. A VQE and its associated buffer cannot be modified or reused until its ECB is posted.

The VQE describes the operation being requested through a pointer and a length. If the length is positive, then the request is for the transfer of a data record of that length. The pointer points to the first data byte of that record. If the length is zero, then the request is for the scheduling of a normal end-of-data-set, and the pointer is not examined. If the length is negative, then the request is to abort the data set in progress, and the pointer is not examined. An abort request takes effect at once, no matter how many records are queued; however, the request is still queued like a data record for purposes of posting. Whenever an end or abort request is posted, the channel is resynchronized, and the operator program can begin a new data set by calling VRBTPUT.

For the card reader stream, there are three elementary events:

1. PUT DATA, in which a data record is to be passed to JES2,

2. PUT END, in which a data set is to be terminated normally,

3. PUT ABORT, in which a data set is to be terminated abnormally.

However, JES2 can also abort a data set, so there are two possible results of each event:

1. POST OK, meaning that the data set is being processed normally.

2. POST ERROR, meaning that the data set is in error, and is being flushed.

From the point of view of the operator program, there are three possible states of the card reader channel, called:

1. SYNCHRONIZED, which is the idle state of the channel. The channel should be allowed to return to this state between data sets in order to simplify error control.

2. SENDING, which is the normal state during data set transmission. During this state, records are being accepted and sent.

3. DRAINING, which is the state between the discovery of an abort condition and the completion of cleanup operations necessary to safely return to state SYNCHRONIZED.

All these things are related by this state diagram:

state=	SYNCHRONIZED	SENDING	DRAINING
event=			
PUT DATA:	POST OK --> state= SENDING POST ERR --> state= DRAINING data set aborted	POST OK --> state= SENDING POST ERR --> state= DRAINING data set aborted	POST OK can't happen POST ERR --> state= DRAINING data was discarded
PUT END:	POST OK can't happen POST ERR --> state= SYNCH'D there was no data set to end	POST OK --> state= SYNCH'D data set was sent POST ERR --> state= SYNCH'D JES2 aborted data set	POST OK can't happen POST ERR --> state= SYNCH'D data set aborted
PUT ABORT:	POST OK can't happen POST ERR --> state= SYNCH'D there was no data set to abort	POST OK --> can't happen POST ERR --> state= SYNCH'D data set aborted	POST OK can't happen POST ERR --> state= SYNCH'D data set aborted

6.2 SENDING MESSAGES

Sending on the keyboard channel is handled more simply than on the other channels, because the keyboard is presumed to have the property that every message is unrelated to every other message. That is, that a "data set" always consists of just one message. Given this presumption, the mechanisms used to delimit data set boundaries are redundant and are not used. When the VQE ECB for a VRBTPUT on the keyboard channel is posted with good status, it means that JES2 has accepted the message. If it is posted in error, it means that the message did not get to JES2 successfully. In neither case is there any direct effect on previous or subsequent messages.

Any call to VRBTPUT with an end-of-data-set or abort-data-set signal will be posted in error and ignored.

6.3 RECEIVING DATA SETS

Once the VRBT is open, the operator program schedules JES2 output data sets (data for the printer, punch, or display) to its buffers by repeatedly calling VRBTGET, passing a VRBT queue element (VQE) which contains an ECB. The program can schedule as many buffers on a channel as it has VQE's to dedicate, with the effect of arbitrarily deep buffering. If multiple buffering is used, the operator program must check the completion status of all buffers in the order that they were queued. Received data is always presented in that order, and the posting of exception conditions that affect the state of the channel is queue-order dependent. The operator program should provide its own ordering mechanism, perhaps through an extended VQE format. It should never attempt to use or interpret the internal chaining fields of the VQE itself. A VQE and its associated buffer cannot be modified or reused until its ECB is posted.

The VQE describes the operation being requested through a pointer and a length. The pointer points to the first data byte of an empty buffer which can accomodate a record of length less than or equal to that indicated by the length field, which must be positive. If and when a record is supplied, VRBTGET will reset the length to the actual length of the record. If the record exceeds the supplied buffer length, it is truncated without comment; therefore, the operator program should never use buffers smaller than the maximum expected record size. JES2 will never send records longer than 255 bytes, so buffers larger than that are never needed.

In addition to scheduling output buffers with VRBTGET, the operator program may have the need to abort an output data set and cause JES2 to hold the data for later retransmission. It will also be required to confirm a normal end-of-data-set signal from JES2. Both these requirements are met by the VRBTACK routine, which communicates a positive or negative status from the VRBT operator program to the

printer or punch channels. VRBTACK should not be called for the display channel; EOD acknowledgements are not required on that channel, and errors on that channel will abort the entire VRBT.

In the first case, the operator program uses VRBTACK to signal an error condition back to JES2, during otherwise normal data transmission, but in the direction opposite data flow. Because of the several levels of mechanism between the operator program and JES2, and because of the possibility of extensive data buffering, this signal will arrive at JES2 at some other point in the data stream than the one at which the operator program sent it. However, the result of the signal will be the transmission of a second error signal back toward the operator program, and this signal, arriving in the data stream, can be synchronized like any other JES2-generated error. Whether an error condition is initiated by JES2 or by the operator program, the operator program must continue to accept buffers queued to VRBTGET until one is posted with the end-of-data-set code.

In the second case, the object is to confirm to JES2, before it purges an output data set, that the entire data contents of that data set have arrived without error, and have been safely disposed of by the end-using process. The mechanism to achieve this is to require that VRBTACK be called by the operator program as a response to any end-of-data-set condition signalled on a buffer scheduled by VRBTGET on the printer or punch (not the display) channels. Until this response is received, no further data will flow in the channel (this also ensures that any negative acknowledgement sent by the operator program does not cross data-set boundaries in transit). The acknowledgement is required regardless of whether the data set is terminating normally or in error. If the termination is already in error mode, the value of the acknowlegement (positive or negative) returned by VRBTACK is immaterial; however, if the data set is terminating normally, a negative VRBTACK is guaranteed to arrive at JES2 in time to cause an error termination. This is the mechanism that prevents such an output data set from being purged.

For the printer, punch, or display data streams, the operator program accepts data, end signals, or abort signals from JES2. It also provides mandatory acknowledgements after an end signal. These are the four synchronous events that determine the state of a VRBT output channel. The operator program can also request a data set abort. This request is asynchronous to the data flow; however, it will eventually cause a synchronous error event. So, from the point of view of the operator program, there are only four elementary synchronous events:

1. GET DATA, in which a data record is received from JES2,

2. GET END, in which a data set is to be terminated normally,

3. GET ABORT, in which a data set is to be terminated abnormally.

4. Mandatory VRBTACK.

There are four states of such a channel, called:

1. SYNCHRONIZED, which is the idle state of the channel. The channel must return to this state between data sets.

2. RECEIVING, which is the normal state during data set transmission. During this state, records are being placed into scheduled buffers.

3. DRAINING, which is the state between the discovery of an abort condition and the completion of cleanup operations necessary to safely return to state SYNCHRONIZED. In this state, any JES2 output records are being flushed, and queued buffers may be being posted with error indicators.

4. CLOSING, which is the state between DRAINING and SYNCHRONIZED during which the two ends of the transmission are brought into perfect synchronism for purposes of error control.

All these things are related by this state diagram:

state=	SYNC'D	RECEIVING	DRAINING	CLOSING
event= GET DATA:	state = RECEIVING	state = RECEIVING	can't happen (will be a GET ABORT)	can't happen
GET END:	state = CLOSING	state = CLOSING	state = CLOSING	can't happen
GET ABORT:	state = DRAINING	state = DRAINING	state = DRAINING	can't happen
VRBTACK:	Ignored	Ignored (see note)	Ignored	state = SYNC'D

NOTE: When VRBTACK is received in the RECEIVING state, and it indicates a negative acknowledgement, an error signal is propagated back to JES. This will cause an eventual GET ABORT event; however, it has no immediate and direct effect on the channel state.

Section 7

SPECIAL SCHEDULING MODE

In special scheduling mode, the calling sequences of all the subroutines except VRBTOPN are extended by one parameter, called the special scheduling return code, or SSRC. The presence of this extra parameter is the only thing that signals the routine that special scheduling is to be used for this call. While it is expected that an operator program will use either all special scheduling or all normal scheduling, nothing requires this to be the case, since the decision is made on a per-call basis.

The special scheduling return code contains two data: the first byte is called the special scheduling flag, and the remaining three bytes are called the special scheduling ECB pointer. Note that, unlike the main ECB of the VQE, which is physically present in the VQE, the special scheduling ECB is located elsewhere, and is pointed to by the SSRC.

On return from a special scheduling request, the special scheduling flag is set to one of three values, chosen for easy testing with the "TM" instructions:

1. flag=X'FF': The request has been scheduled successfully without the need for WAIT or POST (although "fast post" may have been used). Operation will proceed as in normal scheduling mode. In this case, the special scheduling ECB pointer is zero.

2. flag=X'80': The request cannot be scheduled without a WAIT. The special scheduling ECB pointer points to an ECB. The operator program must WAIT (or the equivalent) on this ECB and then repeat the request. The program must not assume that the repeated request will succeed either, although it usually will. No status code is returned in the posted ECB.

3. flag=X'00': The request has been scheduled, but a POST is needed to wake the VRBT task. The special scheduling ECB pointer points to an ECB. The operator program must POST (or the equivalent) this ECB and then proceed as for normal scheduling. The request does not have to be repeated. No particular post code is needed in the POST.

Using the macros expanded in the appendix entitled ASSEMBLER DECLARATIONS, one can test these conditions with the sequence:

```
TM    flag,VSRMSK
BC    VSBOK,ok-processing
```

```
BC    VSBPST,post-processing
BC    VSBWAIT,wait-processing
```

Section 8

VRBT SUBROUTINE CALLS

The VRBT subroutine calls are described below in terms of PL/I CALL statements. The Assembler calls are the same, since the PL/I declarations all use the "OPTIONS(ASSEMBLER)" facility to avoid passing descriptors. The "VL" bit must be set in all calls. See the appendices for the formats of any data fields referenced here by names in all caps.

Some of the VRBT subroutine calls include a data element called a VRBT Queue Element, or VQE. A VQE represents a "pending event". A pending event is created when a VQE is passed to a VRBT entry point, and remains pending until the ECB in the VQE is posted. During the time that the event is pending, the VQE belongs to the VRBT, and it should not be modified or examined by the operator program except to test or WAIT on the ECB. When the event is not pending, the VQE belongs to the operator program, and the VRBT has no knowledge of it.

The operator program initializes the data pointer and data length fields of the VQE according to the entry being called. Before control returns, the ECB will be cleared, so the operator program need never clear a VQE ECB. The chain fields of a VQE are for the internal use of the VRBT, and should not be initialized or altered by the operator program.

8.1 VRBTOPN -- MATERIALIZING THE VRBT

The VRBTOPN routine materializes a VRBT, logs it on to JES2, and primes its channels. Control returns to the caller at once, but the VRBT is not ready for use until an ECB is posted, and then only if the post code is zero. The calling sequence is:

```
CALL VRBTOPN (rmtid, password, openecb,shutecb,
              handle [, modifier-structure ])
```

where:

1. "rmtid" is the terminal ID by which JES2 knows the VRBT to be logged on. This is a small integer.

2. "password" is the 8-character password that JES2 has been told to associate with this VRBT. It is blank if no password has been established.

3. "openecb" is the name of an ECB to be posted when the open operation is complete. VRBTOPN will clear it before returning control, so the operator program need not initialize it in any way. When this ECB is posted, if "shutecb" is not also posted, then the VRBT is logged on to JES2, and will remain so until logged off by a VRBTCLS call.

4. "shutecb" is the name of an ECB to be posted if and when JES2 or some other process vital to the VRBT's existence requests shutdown. If this ECB is ever found posted, then the user should close all channels at the end of their current data sets, and call VRBTCLS. If this ECB is found posted at the same time that "openecb" is found posted, then the terminal was not succesfully materialized. The operator program can only call VRBTCLS to release any allocated resources. If, during normal VRBT operation, JES2 contact is lost, then all active channels will receive an automatic abort-data signal, and this ECB will be posted to ensure that new data transfers are not attempted.

5. "handle" is a pointer variable which receives a value identifying this opening VRBT. This value must be passed unaltered to all subsequent calls until it is invalidated by VRBTCLS. This mechanism permits the VRBT package to be reentrant, and allows an operator program to manage more than one VRBT at a time. The handle is set and is valid on return from the call to VRBTOPN. It is not dependent on the successful materialization of the VRBT, or on the posting of any ECB. Once the operator program is in possesion of a handle, resources are allocated to the VRBT, and VRBTCLS must eventually be called to free them. It is a good idea to call VRBTCLS as a part of abnormal termination of the operator program, when that is possible.

6. "modifier-structure" is a structure containing data that directs VRBTOPN to set up alternate modes of VRBT operation. It is an optional argument which, if provided, can change the behavior of the VRBT. The argument is a data structure that contains several different elements. In order to simplify things, the strucure is defined to consist of fixed-length items, and the first item is a bit mask that determines which of the following items contain data and which are to be ignored. The structure is defined by assembler macro VRBTOPTS, which is listed in an appendix. the data that can be passed include:

   * A bit mask that filters the generation of messages for the VRBTLOG routine. Each bit in the mask corresponds to a class of messages. If the bit is one, the corresponding class of messages will be generated; otherwise, they will not. The correspondence is:

     bit 0 = completed RPL's.
     bit 1 = RU buffers from completed RPL's.
     bit 2 = comments.
     bit 3 = postcodes of important ECB's.
     bit 4 = entry into each internal routine.
     bit 5 = exit from each internal routine.
     bit 6 = postcodes and record contents
             of VQE's.

     If this mask is not enabled, the default is to log completed RPL's, important ECB's, and comments.

   * The 8-character VTAM LUNAME by which JES2 is to be addressed. This datum supports the use of test JES's for VRBT debugging. The default value is "JES2", which selects the production JES2.

   * An 8-character JES2 line passwcrd. The default value for this datum is blanks.

   * An 8-character VTAM ACBNAME password, to be applied to the ACBNAME chosen from the APPLALOC pool. The default value for this datum is blanks. If passwords are to be used, the same one must apply to all ACBNAMES in a pool.

   * A timeout value for the operations involved in meterializing a VRBT. The default is 120 seconds.

   * A debugging switch to arm the debugging dump mechanism. When this bit is set, the VRBT will forget that it is a read-only program, and will store pointers within one of its code sections. This should not be done except to support online debugging activities.

   * An enable bit for ASA carriage control codes of '0' and '-'. If this bit is not set to one, these control codes will be represented as multiple blank lines.

8.2 VRBTCLS -- DESTROYING A VRBT

The VRBTCLS routine does whatever remains to be done to destroy a VRBT and release any resources allocated to it. Depending on the state of the VRBT, this may involve aborting any active data streams, logging off JES2, terminating the VRBT subtask, and freeing main storage. Any queued VQE's are dequeued, but they are not necessarily posted. The calling sequence is:

```
CALL VRBTCLS (handle, ecb [, ssrc ])
```

where:

1. "handle" is the handle value returned by VRBTOPN. After this call is posted complete, this value will be invalid.

2. "ecb" is the name of an ECB to be posted when all operations needed to quiesce the VRBT are complete. It will be cleared before control returns from this call. When it is posted, the low-order 24 bits will receive the corresponding bits of the TCBCMP field of the Task Control Block (TCB) of the VRBT subtask. For normal termination, this will be zero.

3. "ssrc" is a fullword to receive the special scheduling return code. If present, it modifies the behavior of the VRBT. See the section entitled "SPECIAL SCHEDULING MODE".

8.3 VRBTSHT -- REQUESTING SHUTDOWN

The VRBTSHT routine is the symmetric counterpart of the "shutdown ECB". Calling this entry causes the VRBT to request that JES2 not begin any new outbound data sets when the current ones are complete. Because of the usual timing problems, it is possible that a new data set will be begun between the time that the operator program issues this call and the time that the request actually reaches JES2. The operator program should be written in such a way that it will not matter whether this call results in the posting of the "shutdown ecb" or not. The calling sequence is:

```
CALL VRBTSHUT (handle [, ssrc ])
```

where:

1. "handle" is the handle value returned by VRBTOPN.

2. "ssrc" is a fullword to receive the special scheduling return code. If present, it modifies the behavior of the VRBT. See the section entitled "SPECIAL SCHEDULING MODE".

8.4 VRBTPUT -- SENDING DATA TO JES2

The VRBTPUT routine schedules a line of card-reader or keyboard data to be sent to JES2. Control returns at once, but the buffer and VQE may not be reused until an ECB is posted. VRBTPUT also schedules normal and abnormal end-of-data-set signals. The calling sequence is:

```
CALL VRBTPUT (handle, channel, vqe [, ssrc])
```

where:

1. "handle" is the handle value returned by VRBTOPN.

2. "channel" is a either 2 to indicate a remote operator command (keyboard data) or 3 to indicate a jobstream card image (card reader data). If it is anything else, the VQE will be posted "in error", and nothing else will happen.

3. "vqe" is the name of a VQE to represent this pending event The operator program should initialize the data pointer and data length fields to describe the data or exception being passed. Data records will be truncated to 80 bytes for card images and 255 for keyboard input.

   The ECB will be posted when the event is complete. For card-reader data, this will be when the data has been moved out of the operator program's buffer area. For keyboard data, it will be when the message has been accepted by JES2. If the ECB is posted with ADDR(VQEMECB)->VECBCODE¬=0, then an error exists in the channel. For keyboard data, this means that the message was not received by JES2. For card-reader data, it means that the entire data set being sent on this channel is being aborted, and that the channel should be resynchronized. Unless the call that was posted in error was itself a resynchronizing call, the operator program should schedule a normal or error end-of-data-set. Until resynchronization is achieved, all queued data will be flushed and posted with an error code. After resynchronization, the operator program may use VRBTPUT to send another (or to resend the same) data set.

4. "ssrc" is a fullword to receive the special scheduling return code. If present, it modifies the behavior of the VRBT. See the section entitled "SPECIAL SCHEDULING MODE".

8.5 VRBTGET -- RECEIVING DATA FROM JES2

The VRBTGET routine schedules a buffer to receive a line of printer, punch, or display data from JES2. Control returns at once, but the buffer and VQE may not be reused until an ECB is posted. VRBTGET also schedules normal or abnormal end-of-data-set signals. The calling sequence is:

```
CALL VRBTGET (handle, channel, vqe [, ssrc])
```

where:

1. "handle" is handle value returned by VRBTOPN.

2. "channel" is a either 1, requesting an operator message (display data), 4, requesting a line of printer data, or 5, requesting a line of punch data. If it is anything else, the VQE will be posted "in error", and nothing else will happen.

3. "vqe" is the name of a VQE to represent this pending event. The operator program should initialize the data pointer and length. The length must be that of the empty buffer, and should thus be positive and less than 256.

   The ECB will be posted when the event is complete. The low-order 30 bits will receive the usual status code: zero implies that a record has been returned, positive implies that a normal-end-of-data is being returned, *and negative implies an error.* This code is actually redundant with the returned length field, and may not need to be examined.

   When a data record is actually placed in a buffer, the length field will be reduced to that of the actual record. If the record does not fit into the buffer, it will be truncated without comment; therefore, the operator program should never use a buffer smaller than the maximum record that it expects. The VRBT cannot receive data records longer than 255 bytes; records longer than that will be truncated by JES2 without comment. Therefore, the operator program need never use buffers larger than 255 bytes.

   If the returned length field is not positive, it conveys the same information, although in a different form, as the value returned in the posted ECB. If it is negative, then an error exists in the channel, and the entire data set being sent on this channel has been aborted. The operator program should discard this and any subsequent queued buffers that are posted in error, until one indicates a normal end-of-file. For display data, an error means that a message from JES2 to the VRBT operator was not received sucessfully. If the operator program knows what sort of message was expected, then it may be able to send a JES2 command requesting the same data. Otherwise, there is nothing that can be done, and the error should probably be ignored.

If the returned length field is zero, then the current data set has been finalized normally. The operator program can continue to schedule buffers through VRBTGET, but, for the printer and punch (not display) channels, no more data will be delivered until a call has been made to routine VRBTACK, and the next data actually acquired will be for a new data set.

4. "ssrc" is a fullword to receive the special scheduling return code. If present, it modifies the behavior of the VRBT. See the section entitled "SPECIAL SCHEDULING MODE".

8.6 VRBTACK -- PROVIDING ERROR CONTROL

The VRBTACK routine either interrupts an output data set, causing it to terminate in error, or it resynchronizes the two ends of the transmission of a output data set after an end-of-data-set indication. This routine is only used on the printer and punch channels, never on the display channel. The former use is optional, but the latter use is required in order to re-enable data flow on the channel. The calling sequence is:

```
CALL VRBTACK (handle, channel, ack [, ssrc ])
```

where:

1. "handle" is handle value returned by VRBTOPN.

2. "channel" is the channel number of one of the output (JES2 to VRBT) channels.

3. "ack" is a one-bit value indicating:

   * If '1'B, that the data set has been finalized properly, and, unless some other error condition exists, that JES2 can safely purge its copy. This form of the VRBTACK call has no effect unless the channel is in CLOSING state.

   * If '0'B, that the data set has encountered some error. This form of the call can be used at any point in the handling of an output data set to schedule a transmission abort. If used when the channel is in the CLOSING state, it requests that JES2 not delete its copy of the data set just transmitted, but reschedule it for retransmission.

4. "ssrc" is a fullword to receive the special scheduling return code. If present, it modifies the behavior of the VRBT. See the section entitled "SPECIAL SCHEDULING MODE".

Section 9

THE VRBT LOG

For debugging purposes, the VRBT support subroutines can be made to log all VTAM events. This is accomplished through a weak enternal reference to symbol VRBTLOG. If this reference is resolved in the running program, then it will be used in a CALL to dispose of an edited line of log data. If the reference is not resolved, then the log data line is not generated.

The VRBTLOG routine must be an Assembler routine that is capable of being executed under any TCB and under either a PRB or an IRB. Nevertheless, entry into the routine is strictly serialized, even if log data must be lost to accomplish that. Therefore, the routine need not necessarily be reentrant unless more than one VRBT can be materialized by the same load module.

The VRBTLOG calling sequence has deliberately been made incompatible with PL/I. Register 1 is either zero, in which case there is no data associated with this call, or it points directly to a 1-byte length field followed by that many bytes of edited data. Register 13 points to a useable save area. Registers 14 and 15 are as used by "BALR 14,15". Register 0 always contains the same value in its low order three bytes -- the address of a fullword that the VRBTLOG can use for whatever purpose it wishes. For instance, a reentrant routine would store the address of its dynamic storage here. The address is bound to a particular VRBT materialization. The top byte of register 0 contains status bits, as follows:

1. bit 0 = 1 --> this is not the first entry to VRBTLOG. If this bit is 0, the routine should do whatever initialization it requires.

2. bit 1 = 1 --> this will be the last entry to VRBTLOG. If this bit is 1, the routine should do whatever finalization it requires.

3. bit 2 = 1 --> asynchronous log data was lost during the previous entry to VRBTLOG. The routine may ignore this situation if it wishes, but usually it should pass on a lost-data indicator in its own output stream.

Section 10

THE VRBT DEMONSTRATION PROGRAM

The VRBTDEM program is a PL/I main program that is used as a TSO command processor. Its purpose is to exercise or demonstrate the VRBT subroutine package, functioning as a VRBT operator program. It responds to commands from the user, but due to the blocking nature of TSO TGET, it must usually run without a terminal input operation pending. At such times, the user can enter a command through an attention interruption.

So VRBTDEM operates in two modes. In "spin" mode, it concentrates on operating the VRBT, using WAIT when there is no immediate work to do. When the user causes an attention interruption, the VRBT enters "command" mode.

In "command" mode, the VRBT accepts commands from the user's terminal. Between each such command, VRBT operation proceeds as a background activity; however, when there is no immediate work to do, instead of issuing WAIT, the program requests another command. Therefore, little or no work is done until the VRBT again enters "spin" mode.

VRBTDEM drives the reader, printer, and punch channels by connecting them to data sets according to user command. The VRBT operator console is mapped directly onto the user's terminal. Commands may be sent with a SEND command, and messages are displayed on the terminal when they are received.

A VRBTLOG entry is included in VRBTDEM. All log data will be written to a file named VRBTLOG. If no data set has been allocated to this file, no log data will be written. It is permissible to allocate this file to the terminal.

The following commands are recognized and acted on by VRBTDEM.

1. OPEN rmtnn, password

   This command materializes a VRBT using the given terminal id number and password.

2. CLOSE

   This command destroys the VRBT.

3. CONNECT { Reader | PRinter | PUnch } , dsname

This command connects a virtual device to a data set. Only the letters "R", "PR", or "PU" are actually examined. When the connected device is the reader, the data set named by "dsname" is actually read and written to JES2. When end-of-file is reached, the connection is broken. Don't use "*" as the dsname.

When the connected device is the printer or punch, the data set named by "dsname" is set up to receive any output from JES2 on that channel. Data will actually move only when it is sent by JES2. The connection lasts only until the end of the transmission data set. You can use "*" as the dsname.

An attempt to connect an already-connected device will cause an error message, and the command will be ignored.

4. KEYIN text

   This command schedules its entire operand field ("text") as a JES2 command, and sends it down the keyboard channel.

5. RUN

   This command causes VRBTDEM to enter "spin" mode until interrupted by an attention interruption.

6. <attention>

   An attention interruption can be used to interrupt "spin" mode, so that VRBTDEM immediately requests another sequence of commands.

7. <null>

   A null command line can be used to allow background processing to proceed during "command" mode.

8. Other commands

   VRBTDEM supports commands to materialize multiple concurrent VRBTs, and to limit the work done in "spin" mode. These commands are not useful to the casual user, and so are not documented here. They are explained in commentary in the VRBTDEM source file.

Section 11

APPENDIX A -- PL/I DECLARATIONS

```
DECLARE
(VRBTOPN ENTRY (FIXED BIN(31), CHAR(8), FIXED BIN(31),
                FIXED BIN(31), POINTER, *),
 (VRBTGET, VRBTPUT) ENTRY (POINTER, FIXED BIN(15), *),
 VRBTCLS ENTRY (POINTER, FIXED BIN(31)),
 VRBTSHT ENTRY (POINTER),
 VRBTACK ENTRY (POINTER, FIXED BIN(15), BIT(1) ALIGNED))
                            OPTIONS (ASSEMBLER, INTER),
1 VQE BASED,
  2 (VQEFLNK, BQEBLNK) POINTER, /* INTERNAL LINKS      */
  2 VQEMECB FIXED BIN(31),   /* TRANSFER COMPLETE ECB */
  2 VQEDPTR POINTER,         /* ADDR OF BUFFER        */
  2 VQEDLNG FIXED BIN(31),   /* LENGTH BUFFER OR DATA */

1 VRBTCHANNELS STATIC,
  2 (VDISPLAY     INIT(1),   /* CHANNEL NUMBERS       */
     VKEYBOARD    INIT(2),   /* ASSIGNED THE VARIOUS  */
     VREADER      INIT(3),   /* VIRTUAL DEVICES       */
     VPRINTER     INIT(4),
     VPUNCH       INIT(5)) FIXED BIN(15),

1 VRBTRESULTS  STATIC,
  2 (VOK         INIT(0),   /* DATA TRANSFERRED      */
     VERR        INIT(-1),  /* DATA SET ABORT        */
     VEND        INIT(1))   /* END OF DATA SET       */
                         FIXED BIN(15),

1 VRBTACKVALS  STATIC,
  2 (ACKOK       INIT('1'B),/* DATA SET ALL OK       */
     ACKNOK      INIT('0'B))/* DATA SET ABORT        */
                         BIT(1) ALIGNED;
```

Section 12

APPENDIX B -- ASSEMBLER DECLARATIONS

This section lists the macros available and gives a brief explanation of their function. Then it includes the actual unexpanded macro text for each one. These macros are simple, with parameterization extending little beyond labels, so the unexpanded text is expected to be reasonably well self documenting. Available macros are:

VRBTABCD defines the post codes for the "shutdown" ECB.

VRBTACK defines the calling sequence to VRBTACK.

VRBTCLS defines the calling sequence to VRBTCLS.

VRBTEQUS defines the numeric values of the channels and the values associated with using special scheduling.

VRBTGET defines the calling sequence to VRBTGET.

VRBTOPN defines the calling sequence to VRBTOPN.

VRBTOPTS defines the VRBTOPN options structure.

VRBTPARM is a package invoking all the other macros that define calling sequences.

VRBTPUT defines the calling sequence to VRBTPUT.

VRBTSHT defines the calling sequence to VRBTSHT.

VRBTVQE defines the VRBT queue element.

The text of the macros is:

```
          MACRO
&P        VRBTABCD
&P.CLOS   EQU   0         CLOSE REQUESTED BY OPER. PGM.
&P.RSHT   EQU   1         SHUTDOWN REQUESTED BY OPER. PGM.
&P.NOID   EQU   2         THERE ARE NO APPLIDS LEFT
&P.EXIT   EQU   3         UNSUPPORTED EXIT ROUTINE RAN
&P.IRER   EQU   4         CAN'T INIT. INBOUND BRKT. CNTL.
&P.KEYB   EQU   5         ERROR ON KEYBOARD
&P.SHDN   EQU   6         VTAM IS SHUTTING DOWN
&P.OUT    EQU   7         ERROR ON AN OUTPUT CHANNEL
&P.READ   EQU   8         ERROR ON READER CHANNEL
&P.TORS   EQU   9         TIMEOUT IN REQSESS REQUEST
&P.TOBN   EQU   10        TIMEOUT ON BIND
&P.TOOS   EQU   11        TIMEOUT ON OPENSEC
&P.TOSD   EQU   12        TIMEOUT ON START DATA TRAFFIC
&P.TORJ   EQU   13        TIMEOUT ON BIND REJECT
&P.TOES   EQU   14        TIMEOUT IN ENDSESS SEQUENCE
&P.NATT   EQU   15        ATTACH UNSUCCESSFUL
&P.ABND   EQU   16        ABEND -- SEE "CLOSE" ECB FOR WHY
&P.OPEN   EQU   19        UNKNOWN OPEN ERROR
&P.MDEF   EQU   20        NETWORK SEEMS MIS-DEFINED
&P.STOR   EQU   21        TEMPORARY STORAGE SHORTAGE
          MEND
```

```
          MACRO
&V        VRBTACK
&V        DS    0F        PARAMETER LIST TO VRBTACK:
&V.HDL    DS    A         --> F'VRBT HANDLE'
&V.CHAN   DS    A         --> Y(CHANNEL NUMBER)
&V.ENDN   DS    0X'80'    VL BIT FOR NORMAL SCHEDULING
&V.VAL    DS    A         --> BL.1(ACK)
&V.VBIT   EQU   X'80'         BIT 1=1 --> OK,  =0 --> BAD
&V.ENDS   DS    0X'80'    VL BIT FOR SPECIAL SCHEDULING
&V.SSECB  DS    A         --> A(0) TO GET SPEC SCHED ECBAD
          MEND
```

```
          MACRO
&V        VRBTCLS
&V        DS    0F        PARAMETER LIST TO VRBTCLS:
&V.HDL    DS    A         --> F'VRBT HANDLE'
&V.ENDN   DS    0X'80'    VL BIT FOR NORMAL SCHEDULING
&V.ECBC   DS    A         --> A(0) CLOSE COMPLETE ECB
&V.ENDS   DS    0X'80'    VL BIT FOR SPECIAL SCHEDULING
&V.SSECB  DS    A         --> A(0) TO GET SPEC SCHED ECBAD
          MEND
```

```
          MACRO
&V        VRBTEQUS
&V.DSPLY  EQU   1         CHANNEL NUMBER FOR CONSOLE
&V.KEYBD  EQU   2         CHANNEL NUMBER FOR KEYBOARD
&V.READR  EQU   3         CHANNEL NUMBER FOR CARD READER
&V.PRNTR  EQU   4         CHANNEL NUMBER FOR PRINTER
&V.PUNCH  EQU   5         CHANNEL NUMBER FOR PUNCH
          SPACE
&V.SROK   EQU   X'FF'     SS FLAG FOR SCHEDULING COMPLETED
&V.SRWT   EQU   X'80'     SS FLAG FOR WAIT REQUIRED
&V.SRPST  EQU   X'00'     SS FLAG FOR POST REQUIRED
          SPACE
&V.SRMSK  EQU   X'C0'     MASK TO TEST SS FLAG, & BRANCHES:
&V.SBOK   EQU   1  (BO)   SS BRACH FOR SCHEDULING COMPLETED
&V.SBWT   EQU   4  (BM)   SS BRANCH FOR WAIT REQUIRED
&V.SBPST  EQU   8  (BZ)   SS BRANCH FOR POST REQUIRED
          SPACE
&V.NMDVS  EQU   5         NUMBER OF DEVICES/SESSIONS
&V.NMTHS  EQU   6         NUMBER OF THREADS
          MEND
```

```
          MACRO
&V        VRBTGET
&V        DS    0F        PARAMETER LIST TO VRBTGET:
&V.HDL    DS    A         --> F'VRBT HANDLE'
&V.CHAN   DS    A         --> Y(CHANNEL NUMBER)
&V.ENDN   DS    0X'80'    VL BIT FOR NORMAL SCHEDULING
&V.VQE    DS    A         --> VQE DESCRIBING EMPTY BUFFER
&V.ENDS   DS    0X'80'    VL BIT FOR SPECIAL SCHEDULING
&V.SSECB  DS    A         --> A(0) TO GET SPEC SCHED ECBAD
          MEND
```

```
          MACRO
&V        VRBTOPN
&V        DS    0F        PARAMETER LIST TO VRBTOPN:
&V.RMT    DS    A         --> F'RMT #'
&V.PASS   DS    A         --> CL8'JES2 PASSWORD'
&V.ECBO   DS    A         --> A(0) OPEN COMPLETE ECB
&V.ECBS   DS    A         --> A(0) SHUTDOWN ECB
&V.END    DS    0X'80'    VL BIT IF NO OPTIONS.
&V.HDL    DS    A         --> A(0) TO RECEIVE VRBT HANDLE.
&V.ENDO   DS    0X'80'    VL BIT IF OPTIONS.
&V.OPTS   DS    A         --> DEBUGGING OPTIONS.
          MEND
```

```
         MACRO
&L       VRBTOPTS
&L.OPTS  DSECT DESCRIBING THE OPTIONAL DEBUGGING ARGS:
&L.OMAP  DS    BL2      OPTIONAL ARGUMENT MAP:
&L.OMLOG EQU   X'80'       THERE IS A LOG MASK
&L.OMJES EQU   X'40'       THERE IS A JES LUNAME
&L.OMLUPW EQU  X'10'       THERE IS A VTAM LU PASSWORD
&L.OMLPW  EQU  X'08'       THERE IS A LINE PASSWORD
&L.OMRSTO EQU  X'20'       THERE IS A REQSESS TIMEOUT
&L.OMDUMP EQU  X'04'       ENABLE DUMP  (NOT A MAP BIT)
&L.OMSP3  EQU  X'02'       ENABLE ASA CC'S OF "-" AND "0"
         SPACE
&L.OLOG  DS    BL2      LOG MASK
&L.OJES  DS    CL8      JES LUNAME
&L.OLUPW DS    CL8      VTAM LU PASSWORD
&L.OLPW  DS    CL8      LINE PASSWORD
&L.ORSTO DS    FL4      REQSESS TIMEOUT
         MEND


         MACRO
&D       VRBTPARM
&D       DSECT DESCRIBING THE PARAMETERS TO THE VRBT SUBS:
         SPACE
VPO      VRBTOPN
         SPACE
         ORG   &D
VPC      VRBTCLS
         SPACE
         ORG   &D
VPS      VRBTSHT
         SPACE
         ORG   &D
VPG      VRBTGET
         SPACE
         ORG   &D
VPP      VRBTPUT
         SPACE
         ORG   &D
VPA      VRBTACK
         ORG
         MEND
```

```
         MACRO
&V       VRBTPUT
&V       DS    0F        PARAMETER LIST TO VRBTPUT:
&V.HDL   DS    A         --> F'VRBT HANDLE'
&V.CHAN  DS    A         --> Y(CHANNEL NUMBER)
&V.ENDN  DS    0X'80'    VL BIT FOR NORMAL SCHEDULING
&V.VQE   DS    A         --> VQE DESCRIBING DATA BUFFER
&V.ENDS  DS    0X'80'    VL BIT FOR SPECIAL SCHEDULING
&V.SSECB DS    A         --> A(0) TO GET SPEC SCHED ECBAD
         MEND


         MACRO
&V       VRBTSHT
&V       DS    0F        PARAMETER LIST TO VRBTSHT:
&V.ENDN  DS    0X'80'    VL BIT FOR NORMAL SCHEDULING
&V.HDL   DS    A         --> F'VRBT HANDLE'
&V.ENDS  DS    0X'80'    VL BIT FOR SPECIAL SCHEDULING
&V.SSECB DS    A         --> A(0) TO GET SPEC SCHED ECBAD
         MEND


         MACRO
&V       VRBTVQE
&V       DS    0F
&V.FLNK  DS    A         INTERNAL FORWARD CHAIN.
&V.BLNK  DS    A         INTERNAL BACK CHAIN.
&V.MECB  DS    A         MASTER EVENT ECB.
&V.DPTR  DS    A         POINTER TO DATA BUFFER.
&V.DLNG  DS    A         LENGTH OF BUFFER OR DATA.
&V.XTNT  EQU   *-&V      VQE EXTENT.
         MEND
```

Section 13

APPENDIX C -- JES2 INITIALIZATON PARAMETERS

For every distinct VRBT that can be materialized, a declaration must appear in the JES2 initialization data set. The following is a sample of such declarations.

```
LINE2 UNIT=SNA
RMT2    LUTYPE1,CONSOLE,BUFSIZE=512,NUMPU=1
R2.PR1  SEP,LRECL=255
R2.PU1  SEP,LRECL=255
R2.RD1  CLASS=A
LINE3 UNIT=SNA
RMT3    LUTYPE1,CONSOLE,BUFSIZE=512,NUMPU=1
R3.PR1  SEP,LRECL=255
R3.PU1  SEP,LRECL=255
R3.RD1  CLASS=A
```

Section 14

APPENDIX D -- VTAM DECLARATIONS

The VTAM generation deck must include declarations of all the ACBNAMEs that appear in the ACBNAME pool in APPLALOC/APPLFREE. For use with JES2, the following form of declarations is recommended. Note that it is necessary to change the declaration of JES2 itself to include the "PARSESS=YES" option.

```
A03JES2  APPL ACBNAME=JES2,EAS=20,PARSESS=YES,AUTH=(ACQ)
A08ARPAA APPL ACBNAME=JESVRBTA,EAS=5,PARSESS=YES
A08ARPAB APPL ACBNAME=JESVRBTB,EAS=5,PARSESS=YES
A08ARPAC APPL ACBNAME=JESVRBTC,EAS=5,PARSESS=YES
A08ARPAD APPL ACBNAME=JESVRBTD,EAS=5,PARSESS=YES
A08ARPAE APPL ACBNAME=JESVRBTE,EAS=5,PARSESS=YES
A08ARPAF APPL ACBNAME=JESVRBTF,EAS=5,PARSESS=YES
A08ARPAG APPL ACBNAME=JESVRBTG,EAS=5,PARSESS=YES
A08ARPAH APPL ACBNAME=JESVRBTH,EAS=5,PARSESS=YES
A08ARPAI APPL ACBNAME=JESVRBTI,EAS=5,PARSESS=YES
A08ARPAJ APPL ACBNAME=JESVRBTJ,EAS=5,PARSESS=YES
```

Section 15

APPENDIX E -- ALLOCATING VTAM ID'S

Frequently, an ACF/VTAM secondary application program is not tied to a particular Application Identifier (APPLID). In fact, where the same program may be in use by several jobs concurrently, it is imperative that each instance use a different APPLID, i.e. be able to dynamically allocate APPLID's.

To facilitate this purpose, the APPLALOC program defines a group of APPLID pools. Each such pool contains a group of APPLIDS with the same logical attributes. If a secondary application program can operate with one APPLID from a given pool, then it should be able to operate with any APPLID from that pool.

The APPLALOC package allocates and frees dynamically VTAM APPLIDs from various pools. It has two entries: APPLALOC allocates an APPLID, and APPLFREE frees it.

15.1 CALLING SEQUENCES

There are two entries, APPLALOC and APPLFREE. Both are designed to be called from Assembler-language programs by a "BALR 14,15" instruction. Register 13 must point to a standard save area.

15.2 ALLOCATING AN APPLID

To allocate an APPLID, call APPLALOC. On entry, R1 must point to an 8-byte answer place where the 8-character APPLID will be returned. R0 must contain a pool number. On return, R15 contains a return code.

1. 00 -> an APPLID has been returned. The caller has exclusive system-wide control of it until APPLFREE is called, or until the calling task terminates.

2. 04 -> no APPLID has been returned. The requested pool is exhausted.

3. 08 -> no APPLID has been returned.

15.3 FREEING AN APPLID

To free an APPLID, call APPLFREE. On entry, R1 must point to an 8-byte area containing the APPLID to be freed. On return, the calling task will not own the APPLID. No indication is given whether it ever did.

15.4 APPLALOC AND THE OPERATING SYSTEM

APPLALOC is reentrant and read-only. Multiple copies can control the same APPLID pools, so long as the multiple copies are identical. Thus the routine should always be invoked via LINK, and never linkage-edited with a program, since it is expected that the pools will be updated on occasion. To facilitate this, APPLFREE does not check to ensure that the passed APPLID is still in the pool.

While APPLALOC is well-qualified for residency, the expected update frequency and urgency should be taken into account before deciding to make it resident.

15.5 ALLOCATION TECHNIQUE

APPLALOC works by issuing a "SYSTEM" (not "SYSTEMS") ENQ on the selected APPLID RNAME, using QNAME "VTAPPLID", and requesting exclusive control. APPLFREE does the corresponding DEQ.

The selection of an APPLID is done by scanning the internal table for the requested pool, and issuing conditional ENQ macros until either one succeeds or the table is exhausted. In order to ensure that all candidates are tried before giving up, the scan is sequential and circular. In order to minimize the successful search path, the initial scan point is selected randomly, using the low-order bits of the binary time of day.

15.6 DEFINING THE APPLID POOLS

The actual APPLID pools are defined when APPLALOC is assembled, through the use of the APPOOL macro.

Conventionally, pool 0 is a general-purpose pool. Other pool numbers must be in the range 1-254. The assigned numbers need not be contiguous, and the pools need not be defined in any particular order. Pool number 255 is used as an end-of-list indicator, so do not attempt to define a pool with that number.

It is highly reccommended that each pool be made significantly larger than the anticipated need would require, since the instruction path to the determination that a pool is exhausted can be a long one.

A sample pool definition is listed below.

```
APPLALOC CSECT TO HOLD THE ACTUAL NAME POOLS:
         SPACE
BEGPOOLS DS    0F
         APPOOL 0,                                  X
               ARPARM00,ARPARM01,                   X
               ARPARM02,ARPARM03,                   X
               ARPARM04,ARPARM05
         SPACE
         APPOOL 1,                                  X
               ARPARM06,ARPARM07,                   X
               ARPARM08,ARPARM09,                   X
               ARPARM10,ARPARM11,                   X
               ARPARM12,ARPARM12,                   X
               ARPARM14,ARPARM13,                   X
               ARPARM16
         SPACE
         APPOOL ,        TERMINATES POOLS
```

REFERENCES

1 Braden, Network RJS Program Logic Summary, Document s-145, Ofice of Academic Computing, UCLA, October 26, 1973.

2 IBM Corporation, Advanced Communication Function for VTAM: Programming. IBM Document SC27-0449, October, 1980.

3 IBM Corporation OS/VS2 MVS JCL. IBM Document GC28-0692, May, 1979.

4 IBM Corporation, Operator's Library: OS/VS2 MVS JES2 Commands. IBM Document GC23-0007, January, 1979.

5 IBM Corporation, OS/VS2 MVS System Programming Library: JES2. IBM Document GC23-0002, January, 1979.

6 IBM Corporation, OS Assembler H Language. IBM Document GC26-3771, June, 1974.

7 IBM Corporation, OS/VS DOS/VSE VM370 Assembler Language. IBM Document GC33-4010, March, 1979.

8 IBM Corporation, OS PL/1 Checkout and Optimizing Compilers: Language Reference. IBM Document GC33-0009, October, 1976.

9 Ludlam, OAC-JES2 Operators's Guide for Remote Bisynch Terminals. Temporary document, Ofice of Academic Computing, UCLA, September 7, 1980.

PART IV

THE UCLA VIRTUAL LINE TERMINAL SUPPORT PACKAGE

This section is separately available
as UCLA Document TR-25.

Section 1

OVERVIEW OF THE VLT

The UCLA Virtual Line Terminal (VLT) package simulates line-oriented terminals connected to ACF/VTAM (reference 1) application programs such as TSO (reference 9). It is a reentrant subroutine package that creates, manages, and destroys VLT's under the direction of an operator program. The purpose of the support subroutine package is to isolate the VLT operator program from the peculiarities of VTAM process addressing and connection and transmission protocols.

A VLT consists of a keyboard and a line printer, simulating an IBM 3767 buffered typewriter terminal (reference 8). The keyboard is of the unlocked variety. Although the actual transmission of data uses half-duplex contention protocol, collisions are handled by the VLT code, and are not visible to the operator program. Nevertheless, that program must not assume that the two directions of data transfer operate fully independently, and it must never monitor only one channel to the exclusion of the other, since this could cause deadlock.

The operator program uses VLT entries to materialize and destroy a VLT. Once the VLT is successfully materialized, it uses VLT entries to schedule input, output, and attention interruptions.

1.1 CONVERSATIONAL PARTNERS

It is intended that the VLT support package enable any program operating in MVS batch or TSO to drive any VTAM application program that supports the IBM 3767 typewriter terminal, with the most immediate goals being IBM TSO and OBS WYLBUR (reference 10). One of the parameters given the subroutine entry that materializes the VLT describes the VTAM application program by giving its VTAM logical unit name and a protocol type indicator. Both these values should be gotten from the systems programmer who generates VTAM for the installation. Basically, the VLT supports two connection protocols, which we have arbitrarily called type 1 and type 2.

1.1.1 Connection Protocol Type 1

Type 1 protocol is the basic protocol used by most multi-terminal-task time-sharing systems. In this protocol, the VTAM application program is addressed directly by the VLT subroutines, and conversations can begin immediately. It is expected that connections to OBS WYLBUR will use this type of protocol, when OBS announces VTAM support for line-terminal access to WYLBUR.

1.1.2 Connection Protocol Type 2

Type 2 protocol is the protocol used by IBM TSO and similar multi-address-space systems. In this protocol, the program that is addressed directly by the VLT subroutines is merely a router program. That program must create the final conversational partner and must pass the connection to it before conversations can begin. The VLT must cooperate in this connection passing, so it must know that this type of protocol is to be used.

1.2 THE VLT INTERFACE

The VLT package functions as a pseudo-process under the control of the operator program; however, it is not a task. The VLT code can operate in a mode where it never blocks in any form, neither explicit nor implicit, and neither during a normal entry nor during an asynchronous exit routine. For these reasons, the interface between the VLT and its operator program is conceptually divided into two parts: the request interface, which is transaction oriented; and the control interface, which is co-routine oriented.

The request interface defines five types of VLT transactions: open, close, get, put, and attention. Each such transaction is described by a VLT queue element (VQE), and a transacton is initiated when a VQE is passed to the VLT. Every VQE contains an OS Event Control Block (ECB), and the corresponding transaction is completed when that ECB is posted by the VLT. Between the initiation and completion of a VQE-related request, the request is said to be "pending". The operator program can keep any number of VQE's pending, with the effect of arbitrarily deep buffering of input and output requests. VQE's are scheduled by the VLT entries named after the transactions: VLTOPEN, VLTCLOS, VLTGET, VLTPUT, AND VLTATTN.

The control interface is the vehicle for managing the VLT pseudo-process without using tasking or any form of blocking. Essentially, the operator program gives the VLT the address of a single VLT Master ECB, and the VLT causes this one ECB to be posted as the result of the completion of <u>every</u> elementary event that would cause blocking if the VLT were a true task. The operator program agrees that whenever it finds this ECB posted, it will call the VLT package so that the completed elementary event can be handled. The VLT can have many outstanding elementary events at the same time; however, the Master ECB represents them all. The operator program should not be concerned with what the posting of that ECB represents, but it should understand that it <u>never</u> represents the completion of any transaction associated with the request interface.

It does not matter what VLT entry point is used to give control to the VLT package in response to the posting of the Master ECB. Since many elementary internal events must complete in order to complete a single operator request, it will usually be the case that reentry will be through an entry that does not schedule a request. The VLTCONT entry, which represents a "continue" operation, is provided for this case. VLTCONT merely gives control back to the VLT so that it can do whatever it needs to do. Except when using the special OPEN/CLOSE interface (see ENSURING WAIT-FREE OPERATION) it never hurts to call VLTCONT, whether the Master ECB is posted or not. Likewise, it is not necessary to call VLTCONT if one of the entries associated with the request interface could be used instead.

So the operator program has considerable freedom in selecting a mode of integration of the two interfaces. The simplest such programs will use request-scheduling entries only when a request is needed, without regard

to the state of the Master ECB, and will use VLTCONT only when the master ECB is found posted, without regard to whether a request could be scheduled at the same time. In such an program, all pending event ECB's should be scanned after every call to a VLT entry point.

The transaction events represented by the VQE ECB's always complete synchronously to program execution; therefore, the operator program need never block on such ECB's, although it may if it wishes. The elementary internal events represented by the Master ECB usually complete asynchronously; therefore, the operator program must monitor this ECB when it is voluntarily blocked. The servicing of the Master ECB is not critical in real time, so the operator program need not be concerned with monitoring it during involuntary blocks (as during the LOAD or OPEN SVC's).

One cardinal rule must be follwed: if the operator program does wish to block on a VQE ECB, it may not do so without simultaneously blocking on the Master ECB. To disobey this rule is to deadlock the task.

1.3 ERROR HANDLING

The initial version of the VLT will support little or no error recovery. In general, an abnormal postcode from any VLT entry should be interpreted as a non-recoverable error condition affecting the entire VLT. Referring to the definition of macro VLTRTCD in the appendix entitled ASSEMBLER DECLARATIONS, you will see that codes "OK" and "PURGE" (0 and 1) are "normal", while all greater codes are "abnormal". Once an abnormal code has been received, the operator program should probably call VLTCLOS to finalize the VLT.

1.4 TASKING CONSTRAINTS

Theoretically, it is possible to call the VLT entries from various tasks; however, this is strongly discouraged. The various VLT entries are not protected from multiple concurrent entry, and so multi-task calls must be strictly serialized by the operator program. There is one definite requirement. the VTAM ACB must be opened and closed by the same task. This is easily managed if the operator program uses the special non-blocking OPEN/CLOSE interface (described later), and issues OPEN and CLOSE itself. However, if the operator program allows the VLT to issue OPEN and CLOSE, it cannot tell reliably which call to which routine will actually cause the critical operation to be issued. In such a case, multi-task calls are dangerous.

1.5 ENSURING WAIT-FREE OPERATION

The VLT itself does not require a wait-free environment; however, many of its using programs may have that requirement, so mechanisms are provided to support it. There are three operations needed by the VLT which can cause implicit blocks. These are: OPEN for the VTAM ACB, CLOSE for the VTAM ACB, and LINK to the APPLALOC or APPLFREE external entries. If the operator program does not mind these blocks, the VLT will allow them to occur. Otherwise, the operator program must take measures to prevent them.

To prevent blocking during LINK, the operator program should ensure that entries APPLALOC and APPLFREE are in virtual storage and available to the current task whenever the VLT subroutines are being used. Because these entries constitute a system-wide pool of ACBNAMEs that is subject to update, there should only be one copy of them in the system; therefore, they should not be linkage-edited into any load module. An operator program with critical blocking requirements will have a mechanism for issuing LOAD and DELETE without blocking the critical task.

To prevent blocking during OPEN and CLOSE, the operator program can define, in the VLTOPEN call, a special OPEN/CLOSE interface to the VLT. This interface is largely independent of the request and control interfaces, and works this way: The operator program defines a one-word OPEN/CLOSE parameter list (which need not be initialized) and an OPEN/CLOSE ECB. Whenever the VLT requires OPEN or CLOSE, it will initialize the OPEN/CLOSE parameter list, and it will post the ECB with a code indicating whether OPEN or CLOSE is desired. The master ECB is NOT posted concurrently. It is the responsibility of the operator program to issue the OPEN or CLOSE macro-instruction and to complete or abandon the operation before reentering the VLT with this handle via any entry point (even the "null" entry VLTCONT). If the VLT is reentered with a pending OPEN incomplete, the VLTOPN request will fail. If it is reentered with a pending CLOSE incomplete, it will issue CLOSE itself, thus incurring uncontrolled blocks for the operator program. The VLT is rather adamant about getting its ACB closed in one way or another.

Actually, the operator program need not monitor the OPEN/CLOSE ECB at all times. An OPEN request is only possible while a VLTOPN VQE is pending, and a CLOSE request is only possible while a VLTCLS VQE is pending. The VLT subroutines will not alter the ECB or parameter list except at the moments of requesting OPEN or CLOSE processing; otherwise, those words are considered to belong to the operator program.

1.6 THE VLT PACKAGE ENVIRONMENT

All VLT entries are PL/I (optimizing compiler) compatible (reference 7) through the "OPTIONS (ASSEMBLER)" facility. Canned declarations are provided for both the PL/I and the Assembler caller. The PL/I declarations treat ECB's as FIXED BIN (31) data.

The VLT support subroutines are contained in two control sections

1. VLTSUBS contains the bulk of the code. It is about 5600 bytes long.

2. VLTTABS contains code and tables to support the formatting of VTAM RPL's for the VLTLOG function. It can be placed in such a way by the linkage edit that it will remain paged out when logging is not enabled. If you can guarantee that logging will never be enabled, it can be deleted from the load module. It is just under 4K bytes long. This section is identical to the one of the same name which is a part of the UCLA Virtual Remote Batch Terminal (VRBT) support package.

In addition to the static storage occupied by these control sections, each materialization of a VLT will acquire approximately 3K bytes of working storage from subpool 101. This storage is released when VLTCLOS is posted complete.

1.7 VLT RECORD FORMATS

All data records handled by VLTGET and VLTPUT are formatted as segments of an interactive terminal control stream. The actual stream characteristics are determined by the application program on the other end. These things are known about TSO's requirements on the stream:

1. The inbound stream can pack as many lines as will fit into a buffer. Each is terminated by the EBCDIC "newline" character.

2. A line cannot be split accross inbound buffers.

3. The outbound stream can pack multiple lines into a buffer, and may also split lines accross buffers. In any case, lines are terminated by "newline".

The VLT appears to the system to be an IBM 3767 buffered typewriter terminal. Thus the maximum buffer size that can be transmitted in either direction is 256 bytes. An attempt to send more than this will cause an unrecoverable error on the VLT keyboard.

The VLT support subroutines do not examine, interpret, or transform data in any way.

1.8 KEYBOARD MANAGEMENT

The VLT is, for all practical purposes, an unlocked-keyboard device. It is never illegal to call VLTPUT to put a buffer to the keyboard. Most operator programs need not concern themselves beyond that fact.

However, there are moments when an output buffer is "chained" to a following buffer. This situation can be likened to a locked keyboard, since the VLT will hold any pending keyboard buffer until the "chained" buffer has been received. Presumably, chained buffers are all ready for transmission at once, so that receiving subsequent buffers will not cause delay. For the occasional program which may need to make use of this fact, the VLT distinguishes, when it POSTs a VLTGET request complete, whether a subsequent buffer is chained to the one being delivered. Programs which have no need for this information should treat the two cases the same.

Section 2

VLT SUBROUTINE CALLS

The VLT subroutine calls are described below in terms of PL/I CALL statements. The Assembler calls are the same, since the PL/I declarations all use the "OPTIONS(ASSEMBLER)" facility to avoid passing descriptors. The "VL" bit must be set in the VLTOPEN call; however, for compatibility with possible future versions, it is wise to set it in all calls.

When a call includes a VQE, it creates a pending event. The operator program should usually preset the data pointer and data length fields of the VQE before creating the pending event. The called entry will clear the ECB before control returns. The chain fields of the VQE are for the internal use of the VLT, and need not be initialized or examined by the operator program. While an event is pending, its VQE belongs to the VLT, and the operator program should not alter it or examine it in any way other than to test or wait on its ECB. At all times when it does not represent a pending event, the VQE belongs to the operator program, and the VLT is not aware of its existence.

2.1 VLTOPEN -- MATERIALIZING THE VLT

The VLTOPEN routine materializes a VLT, connects it to the designated application program, and optionally passes a LOGON string to that program. The handling of LOGON strings depends on the application program, but in any case the string is formatted as the application program expects it, and not as a string for processing by VTAM Unformatted System Services (USS). In VTAM terms, this means that it is the contents of the DATA field of a standard VTAM LOGON.

In the specific case of TSO, you have two options: you can pass a standard TSO LOGON command, with or without the verb "LOGON," or you can pass a null string. In the latter case, TSO will act as if it received the single word "LOGON," and it will prompt the user for his logonid and password. In no case is it necessary for the TSO LOGON to be completed for the VLTOPN to complete, since TSO may want to use the opened VLT to acquire parameters necessary to complete the LOGON.

The calling sequence is:

```
CALL VLTOPEN (handle, vqe, partner, masterecb,
              [, modifiers ])
```

where:

1. "handle" is a pointer variable which receives a value identifying this opening VLT. This value must be passed unaltered to all subsequent calls until it is invalidated by VLTCLOS. This mechanism permits the VLT package to be reentrant, and allows an operator program to manage more than one VLT at a time. The handle is set and is valid on return from the call to VLTOPEN. It is not dependent on the successful materialization of the VLT, or on the posting of any ECB. Once you are in possession of such a handle, storage resources are allocated to the VLT, and VLTCLOS must eventually be called to free them.

2. "vqe" is the name of a VQE to represent this pending request. The data pointer and length fields should be initialized to describe the LOGON string. If this is to be null, then the data length field should be set to zero. This string is not considered to be terminal input, so it need not end with a "newline" character.

   The ECB will be cleared before control returns, and it will be posted when the open operation is complete. If the postcode portion of the posted ECB contains a non-zero value, the open failed, and the operator program must call VLTCLOS to release allocated resources. The possible postcodes are listed in an appendix.

   The chain fields of the VQE are for the internal use of the VLT, and should not be initialized or examined by the operator program.

3. "partner" is a structure describing the VTAM application program to be addressed. It is nine bytes long, and consists of a 1-byte connection protocol number followed by the 8-character VTAM LUNAME of the program. The protocol number is examined only in its lower four bits, so it can be either a binary number or its EBCDIC equivalent, at the convenience of the operator program. Initially, only connection protocols 1 and 2 are supported. Examples of valid "partner" strings are:

   ```
   CL9'1WYL     '
   CL9'2TSO     '
   ```

4. "masterecb" is the name of an ECB to be posted when the VLT requires control of the task. It will be cleared and posted by VLT code, so the caller need never initialize it or set it in any way.

5. "modifiers" is a structure containing data that directs VLTOPEN to set up alternate modes of VLT operation. It is an optional argument which, if provided, can change the behavior of the VLT. The argument is a data structure that contains several different

elements. In order to simplify things, the structure is defined to consist of fixed-length items, and the first item is a bit mask that determines which of the following items contain data and which are to be ignored. The structure is defined by assembler macro VLTOPTS, which is listed in an appendix. The data that can be passed include:

* A bit mask that filters the generation of messages for the VLTLOG routine. Each bit in the mask corresponds to a class of messages. If the bit is one, the corresponding messages will be generated; otherwise, they will not. The correspondence is:

    bit 0 = completed VTAM RPL's.
    bit 1 = RU buffers from completed RPL's.
    bit 2 = comments.
    bit 3 = the postcodes of important ECB's.
    bit 4 = entry into each internal routine.
    bit 5 = exit from each internal routine.

  If this mask is not supplied, the default is to log RPL's and comments.

* An 8-character VTAM logon mode name. If this is not supplied, the IBM logon mode entry named INTERACT will be used. This is the entry that supports the IBM 3767.

* A fullword parameter list and ECB to support the special OPEN/CLOSE interface. If this interface is not defined here, the VLT will issue OPEN and CLOSE internally, causing uncontrolled blocks in the operator program's task.

2.2 VLTCLOS -- DESTROYING THE VLT

The VLTCLOS routine does whatever remains to be done to destroy a VLT and release any resources allocated to it. Depending on the state of the VLT, this may involve aborting any active data streams, disconnecting from the VTAM application program, disconnecting from VTAM, and freeing main storage.

VLTCLOS will purge any other pending VQE's of any kind, posting their ECB's with the "PURGE" postcode.

The calling sequence is:

```
CALL VLTCLOS (handle, vqe)
```

where:

1. "handle" is the handle value returned by VLTOPEN. After this request is posted complete, this value will be invalid.

2. "vqe" is the name of a VQE to represent this pending request. The data pointer and length fields are not used; however, for future compatibility, it is wise to initialize the length field to zero.

   The ECB will be cleared before control returns, and it will be posted when the close operation is complete. Until it is posted, the operator program must continue to monitor the Master ECB, and to call VLTCONT. After it is posted, VLTCONT must not be called.

   The chain fields of the VQE are for the internal use of the VLT, and should not be initialized or examined by the operator program.

2.3 VLTPUT -- SENDING DATA THROUGH THE VIRTUAL KEYBOARD

The VLTPUT routine schedules a buffer of keyboard data to be sent to the VTAM application program. Control returns at once, but the buffer and VQE may not be reused until the ECB is posted. VLTPUT may be called at any time there is a handle defined. If the VLT is not yet fully open, the request will remain pending until it is.

The operator program can schedule as many pending VQE's through this routine as it wishes. They will always be posted in the order queued, so the operator program should keep track of the oldest such VQE for purposes of testing completion. Of course, the operator program must also be prepared to ignore queued requests that are completed with the "PURGE" postcode.

The calling sequence is:

```
CALL VLTPUT (handle, vqe)
```

where:

1. "handle" is the handle value returned by VLTOPEN.

2. "vqe" is the name of a VQE to represent this pending data transfer. The caller should initialize the data pointer and data length fields to describe the data being passed. This cannot exceed 256 bytes.

   The ECB will be cleared before control returns, and it will be posted when the buffer can be reused. Possible postcodes are listed in an appendix.

   The chain fields of the VQE are for the internal use of the VLT, and should not be initialized nor examined by the caller.

2.4 VLTGET -- RECEIVING DATA FOR THE VIRTUAL PRINTER

The VLTGET routine schedules a buffer to receive printer data from the VTAM application program. Control returns at once, but the buffer and VQE may not be reused until an ECB is posted. VLTGET may be called at any time there is a handle defined. If the VLT is not yet fully open, the request will remain pending until it is.

The operator program can schedule as many pending VQE's through this routine as it wishes. They will always be posted in the order queued, so the operator program should keep track of the oldest such VQE for purposes of testing completion. Of course, the operator program must also be prepared to ignore queued requests that are completed with the "PURGE" postcode.

The calling sequence is:

```
CALL VLTGET (handle, vqe)
```

where:

1. "handle" is handle value returned by VLTOPEN.

2. "vqe" is the name of a VQE to represent this pending data transfer. The caller should initialize the data pointer and length. The length must be that of the empty buffer, and should thus be positive and less than 256.

   The ECB will be cleared before control returns, and it will be posted when the data transfer is complete. The postcodes are defined by macro VLTRTCD, which is listed in Appendix B. There are two kinds of postcodes: "normal" and "abnormal". If the postcode is "abnormal", then an unrecoverable error has occurred, and data is not available. There are three "normal" postcodes:

   * "OK" -- the operation completed normally in every way.

   * "OKCH" -- the operation completed normally, and there is more output chained to this buffer. Most programs will treat this the same as "OK".

   * "PURGE" -- the operation was rescinded by the VLTCLOS, VLTPRG, or VLTATTN routines.

When data is actually placed in a buffer, the length field will be reduced to that of the actual data. Excess data will simply be delivered in a subsequent buffer.

The chain fields of the VQE are for the internal use of the VLT, and should not be initialized nor examined by the caller.

2.5 VLTATTN -- PRESSING THE VIRTUAL BREAK KEY

The VLTATTN routine schedules a VTAM SIGNAL command to the VTAM application program, including four bytes of binary "signal data". Normally, this is equivalent to pressing the "break" or "ATTN" key; however, its actual interpretation is up to the VTAM application program. In the specific case of TSO, it is believed, though at this writing it is not verified, that the signal data is interpreted as a count of simultaneous attention interruptions, with a count of 1 representing the usual case of a single pressing of the break key.

In keeping with its usual use, VLTATTN will purge any pending VLTPUT requests, posting their VQE ECB's with the "PURGE" postcode. On the VLT level, VLTATTN has no direct effect on the outbound (printer) data stream.

VLTATTN may be called at any time there is a handle defined; however, if VLTOPEN or VLTCLOS is pending, the request will be rejected with a postcode indicating a harmless request reject that does not affect VLT operation. In that case, pending VLTPUT VQE's will not be purged.

The calling sequence is:

```
CALL VLTATTN (handle, vqe)
```

where:

1. "handle" is the handle value returned by VLTOPEN.

2. "vqe" is the name of a VQE to represent this pending request. The caller should initialize the data length field to the value of the signal data, normally binary 1. The data pointer field is not presently used.

   The ECB will be cleared before control returns, and it will be posted when the VQE can be reused. Possible postcodes are listed in an appendix.

   The chain fields of the VQE are for the internal use of the VLT, and should not be initialized nor examined by the caller.

2.6 VLTCONT -- SHARING THE TASK WITH THE VLT

The VLTCONT routine merely returns control of the operator program's task to the VLT pseudo-process, so that, if it is logically unblocked, it can proceed. Normally this entry is called as a result of the operator program's having found the VLT Master ECB posted. This entry does not schedule any pending request; however, it may complete one or more previously pending requests. So, although there is no ECB associated with this call, on return from it a scan of pending VQE's might be wise. Unnecessary calls to VLTCONT do no harm.

The calling sequence is:

```
CALL VLTCONT (handle)
```

where "handle" is handle value returned by VLTOPEN.

2.7 VLTPRG -- PURGING PENDING EVENTS

The VLTPRG routine purges pending VQE's from either the VLTGET queue or the VLTPUT queue. Purged VQE's will be posted with the "PURGE" postcode. Notice that the queues contain only VQE's that have been scheduled by the operator program, but have not yet been dequeued and made the objects of active VTAM operations. "Active" buffers of the latter type cannot be purged by this call.

VLTPRG completes at once, and does not use a VQE or ECB. The calling sequence is:

```
CALL VLTPRG (handle, options)
```

where

1. "handle" is handle value returned by VLTOPEN.

2. "options" is a 2-bit string, aligned on a byte boundary, specifying which queues are to be purged. If the first bit is 1, then the VLTGET queue is purged. If the second bit is 1, then the VLTPUT queue is purged. All combinations of these two bits are valid.

2.8 VLTKNT -- COUNTING PENDING EVENTS

The VLTKNT routine counts pending VQE's on the VLTGET queue and the VLTPUT queue. Notice that the queues contain only VQE's that have been scheduled by the operator program, but have not yet been dequeued and made the objects of active VTAM operations. "Active" buffers of the latter type cannot be counted by this call.

VLTKNT completes at once, and does not use a VQE or ECB. The calling sequence is:

```
CALL VLTPRG (handle, counts)
```

where

1. "handle" is handle value returned by VLTOPEN.

2. "counts" is a sequence of two halfwords to receive the counts. The first halfword receives the count from the VLTGET queue, and the second receives that from the VLTPUT queue.

Section 3

THE VLT LOG

For debugging purposes, the VLT support subroutines can be made to log all VTAM events. This is accomplished through a weak enternal reference to symbol VLTLOG. If this reference is resolved in the running program, then it will be used in a CALL to dispose of an edited line of log data. If the reference is not resolved, then the log data line is not generated.

The VLTLOG routine must be an Assembler routine that is capable of being executed under any TCB and under either a PRB or an IRB. Nevertheless, entry into the routine is strictly serialized, even if log data must be lost to accomplish that. Therefore, the routine need not necessarily be reentrant, unless, of course, more than one VLT is to be materialized at a time.

When determining whether to use a VLTLOG routine, remember that it will execute under the task of the operator program, that any blocks in it will delay the operator program, and that it does not have access to the Master-ECB mechanism for preventing blocks. If blocking must be prevented, then the VLTLOG mechanism must not be used.

The VLTLOG calling sequence has deliberately been made incompatible with PL/I. Register 1 is either zero, in which case there is no data associated with this call, or it points directly to a 1-byte length field followed by that many bytes of edited data. Register 13 points to a useable save area. Registers 14 and 15 are as used by "BALR 14,15". Register 0 always contains the same value in its low order three bytes -- the address of a fullword that the VLTLOG can use for whatever purpose it wishes. For instance, a reentrant routine would store the address of its dynamic storage here. The address is bound to a particular VLT materialization. The top byte of register 0 contains status bits, as follows:

* bit 0 = 1 --> this is not the first entry to VLTLOG. If this bit is 0, the routine should do whatever initialization it requires.

* bit 1 = 1 --> this will be the last entry to VLTLOG. If this bit is 1, the routine should do whatever finalization it requires.

* bit 2 = 1 --> asynchronous log data was lost during the previous entry to VLTLOG. The routine may ignore this

situation if it wishes, but usually it should pass on a lost-data indicator in its own output stream.

Section 4

THE VLT DEMONSTRATION PROGRAM

The VLTDEM program is a PL/I main program that is used as a TSO command processor. Its purpose is to exercise or demonstrate the VLT subroutine package, functioning as a VLT operator program communicating with TSO.

VLTDEM relays input from its user, but due to the blocking nature of TSO TGET, it must run without a terminal input operation pending, so input must be entered after an attention interruption. Input consisting of a single "*" results in an attention interruption being passed on to TSO.

Output received from TSO is printed on the user's terminal preceeded by the string "*****".

VLTDEM includes a VLTLOG routine that issues TPUT's to the user's terminal. The logging mask, along with some other options, is requested by VLTDEM before it opens its VLT. Data that are requested of the VLTDEM user include the remote VTAM application program's VTAM LUNAME, an optional LOGON string, and a VLTLOG mask. The mask is in the form of a character string with no real format. The presence of particular letters in the string will cause matching bits in the mask to be set. The presence of other characters is ignored. The correspondence is:

R = completed VTAM RPL's.
B = completed VTAM RU buffers.
C = comments.
P = ECB posts.
E = internal routine entries.
X = internal routine exits.

Section 5

APPENDIX A -- PL/I DECLARATIONS

```
DECLARE
 (VLTOPEN ENTRY (POINTER, *, CHAR(9), POINTER, *),
  (VLTGET, VLTPUT, VLTATTN, VLTCLOS) ENTRY (POINTER, *),
  VLTPRG  ENTRY (POINTER, BIT(2) ALIGNED),
  VLTKNT  ENTRY (POINTER, *),
  VLTCONT ENTRY (POINTER))    OPTIONS (ASSEMBLER, INTER),
 1 VQE BASED,
   2 (VQEFLNK, BQEBLNK) Ptr, /* INTERNAL LINK FIELDS  */
   2 VQEMECB POINTER,        /* TRANSFER COMPLETE ECB */
   2 VQEDPTR POINTER,        /* ADDR OF BUFFER        */
   2 VQEDLNG FIXED BIN(31),  /* LENGTH BUFFER OR DATA */
 1 VECB BASED,               /* ECB FORMAT:           */
   2 VECBCTL BIT(16),        /*   CONTROL FIELDS      */
   2 VECBCODE FIXED BIN(15), /*   STATUS CODE         */
 1 VOPTS BASED,
   2 VOMASK,
     3 (VOMLOG,              /* THERE IS A LOG MASK   */
        VOMMOD,              /* THERE IS A MODE NAME  */
        VOMOCI,              /* THERE IS AN O/C INTFC */
        VOMSPARE (13)) BIT (1) UNALIGNED,
   2 VOLOG   bit (16),       /* LOG MASK              */
   2 VOMOD   CHAR (8),       /* LOGON MODE NAME       */
   2 VOOCI,                  /* OPEN/CLOSE INTERFACE: */
     3 VOOCIECB POINTER,     /*    ECB                */
     3 VOOCIPTR POINTER;     /*    PTR TO ACB         */
```

Section 6

APPENDIX B -- ASSEMBLER DECLARATIONS

This section lists the macros available and gives a brief explanation of their function. Then it includes the actual unexpanded macro text for each one. These macros are simple, with parameterization extending little beyond labels, so the unexpanded text is expected to be reasonably well self documenting. Available macros are:

VLTATTN defines the calling sequence to VLTATTN.
VLTCLOS defines the calling sequence to VLTCLOS.
VLTCONT defines the calling sequence to VLTCONT.
VLTGET defines the calling sequence to VLTGET.
VLTKNT defines the calling sequence to VLTKNT.
VLTOPEN defines the calling sequence to VLTOPEN.
VLTOPTS defines the VLTOPEN options structure.
VLTPARM is a package invoking all the other macros
that define calling sequences.
VLTPRG defines the calling sequence to VLTPRG.
VLTPUT defines the calling sequence to VLTPUT.
VLTRTCD defines the postcodes for VQE ECB's.
VLTVQE defines the VLT queue element.

The text of the macros is:

```
         MACRO
&V       VLTATTN
&V       DS    0F       PARAMETER LIST TO VLTATTN:
&V.HDL   DS    A        --> HANDLE FROM VLTOPEN.
&V.END   DS    0X'80'   VL BIT.
&V.VQE   DS    A        --> VQE WITH ATTENTION ECB.
         MEND


         MACRO
&V       VLTCLOS
&V       DS    0F       PARAMETER LIST TO VLTCLOS:
&V.HDL   DS    A        --> HANDLE FROM VLTOPEN.
&V.END   DS    0X'80'   VL BIT.
&V.VQE   DS    A        --> VQE WITH CLOSE ECB.
         MEND
```

```
         MACRO
&V       VLTCONT
&V       DS    0F        PARAMETER LIST TO VLTCONT:
&V.END   DS    0X'80'    VL BIT.
&V.HDL   DS    A         --> HANDLE FROM VLTOPEN.
         MEND


         MACRO
&V       VLTGET
&V       DS    0F        PARAMETER LIST TO VLTGET:
&V.HDL   DS    A         --> HANDLE FROM VLTOPEN.
&V.END   DS    0X'80'    VL BIT.
&V.VQE   DS    A         --> VQE DESCRIBING BUFFER.
         MEND


         MACRO
&V       VLTKNT
&V       DS    0F        PARAMETER LIST TO VLTKNT:
&V.HDL   DS    A         --> HANDLE FROM VLTOPEN.
&V.END   DS    0X'80'    VL BIT.
&V.OPTS  DS    A         --> 3H FOR OUTBOUND, INBOUND,
*                            ATTENTION COUNTS.
         MEND


         MACRO
&V       VLTOPEN
&V       DS    0F        PARAMETER LIST TO VLTOPEN:
&V.HDL   DS    A         --> A(0) TO RECEIVE HANDLE.
&V.VQE   DS    A         --> VQE WITH OPEN ECB & LOGON DATA.
&V.PART  DS    A         --> PARTNER DESCRIPTOR.
&V.END   DS    0X'80'    VL BIT WHEN NO OPTIONS PRESENT.
&V.MECB  DS    A         --> MASTER EVENT ECB.
&V.ENDO  DS    0X'80'    VL BIT WHEN OPTIONS ARE PRESENT.
&V.OPTS  DS    A         --> OPTIONS STRUCTURE.
         MEND
```

```
         MACRO
&L       VLTOPTS &DSECT=YES
         AIF   ('&DSECT' NE 'YES').NODSCT
&L.OPTS  DSECT DESCRIBING THE OPTIONAL ARGUMENTS:
         AGO   .COM
.NODSCT  ANOP
&L       DS    0F        THE OPTIONAL ARGUMENTS
.COM     ANOP
&L.OMAP  DS    BL2       OPTIONAL ARGUMENT MAP:
&L.OMLOG EQU   X'80'       THERE IS A LOG MASK
&L.OMMOD EQU   X'40'       THERE IS AN LOGON MODE NAME
&L.OMOPN EQU   X'20'       THERE IS AN OPEN/CLOSE INTERFACE
         SPACE
&L.OLOG  DS    BL2       LOG MASK
&L.OMODE DS    CL8       LOGON MODE NAME
&L.OOCI  DS    0XL8      OPEN/CLOSE INTERFACE:
&L.OOCE  DS    F           PTR TO ECB TO POST TO REQUEST
&L.OOPEN EQU   0           CALLER TO:  OPEN  THE ACB, OR
&L.OCLOS EQU   1                       CLOSE THE ACB.
&L.OOCL  DS    A           ADDRESS OF OPEN/CLOSE PARAM LIST.
         MEND


         MACRO
&D       VLTPARM
&D       DSECT DESCRIBING PARAMETERS TO VLT SUBROUTINES:
         SPACE
VPO      VLTOPEN
         SPACE
         ORG   &D
VPC      VLTCLOS
         SPACE
         ORG   &D
VPK      VLTKNT
         SPACE
         ORG   &D
VPG      VLTGET
         SPACE
         ORG   &D
VPPR     VLTPRG
         SPACE
         ORG   &D
VPP      VLTPUT
         SPACE
         ORG   &D
VPA      VLTATTN
         ORG
         ORG   &D
VPCO     VLTCONT
         ORG
         MEND
```

```
         MACRO
&V       VLTPRG &HEADER=YES
         AIF    ('&HEADER' NE 'YES').NOHDR
&V       DS     0F        PARAMETER LIST TO VLTPRG:
.NOHDR   ANOP
&V.HDL   DS     A         --> HANDLE FROM VLTOPEN.
&V.END   DS     0X'80'    VL BIT.
&V.OPTS  DS     A         --> BL.2:
&V.PGET  EQU    X'80'         PURGE OUTBOUND STREAM
&V.PPUT  EQU    X'40'         PURGE INBOUND STREAM
&V.PATTN EQU    X'20'         PURGE ATTENTION STREAM
         MEND


         MACRO
&V       VLTPUT
&V       DS     0F        PARAMETER LIST TO VLTPUT:
&V.HDL   DS     A         --> HANDLE FROM VLTOPEN.
&V.END   DS     0X'80'    VL BIT.
&V.VQE   DS     A         --> VQE DESCRIBING DATA.
         MEND


         MACRO
&P       VLTRTCD ,     POST CODES FROM VLT SUBROUTINES:
*              "NORMAL" VLT RETURN CODES:
&P.OK    EQU    0         OPERATION COMPLETED NORMALLY
&P.OKCH  EQU    1         NORMAL WITH CHAINED BUFFER
&P.PURGE EQU    2         BUFFER PURGED BY ANOTHER ENTRY
*              "ABNORMAL" VLT RETURN CODES:
&P.PERR  EQU    3         "I/O" ERROR IN A "PUT" OPERATION
&P.GERR  EQU    4         "I/O" ERROR IN A "GET" OPERATION
&P.AERR  EQU    5         THERE ARE NO APPLIDS LEFT,
*                         OR PASSED-IN APPLID IS BAD.
&P.RSERR EQU    6         ERROR IN REQSESS REQUEST
&P.BNERR EQU    7         BAD BIND IMAGE
&P.SGERR EQU    8         ERROR IN SIGNAL
&P.SLERR EQU    9         ERROR IN SETLOGON
&P.OSERR EQU    10        ERROR IN OPENSEC
&P.RTERR EQU    11        ERROR IN RESET
&P.OPERR EQU    12        CANNOT OPEN ACB:  UNKNOWN REASON
&P.OPMDEF EQU   13                NETWORK SEEMS MISDEFINED
&P.OPSTOR EQU   14                STORAGE SHORTAGE
&P.OPSHDN EQU   15                VTAM SHUTTING DOWN
&P.TSERR EQU    16        ERROR IN TRMSESS
&P.QERR  EQU    17        VQE QUEUEING ERROR
&P.TOERR EQU    20        RESERVED FOR EXT OPEN TIME-OUT ERR
         MEND
```

```
         MACRO
&V       VLTVQE &HEADER=YES
         AIF    ('&HEADER' NE 'YES').NOHDR
&V       DS     0F
.NOHDR   ANOP
&V.FLNK  DS     A                    INTERNAL FORWARD CHAIN.
&V.BLNK  DS     A                    INTERNAL BACK CHAIN.
&V.RECB  DS     A                    REQUEST ECB.
&V.DPTR  DS     A                    POINTER TO DATA BUFFER.
&V.DLNG  DS     A                    LNG OF BUFFER OR DATA.
&V.XTNT  EQU    *-&V                 VQE EXTENT.
         MEND
```

Section 7

APPENDIX C -- VTAM DECLARATIONS

The VTAM generation deck must include declarations of all the ACBNAMEs that appear in the ACBNAME pool in APPLALOC/APPLFREE. For use with TSO, the following form of declarations is reccommended.

```
A08ARPAA APPL ACBNAME=TSOVLTA
A08ARPAB APPL ACBNAME=TSOVLTB
A08ARPAC APPL ACBNAME=TSOVLTC
A08ARPAD APPL ACBNAME=TSOVLTD
A08ARPAE APPL ACBNAME=TSOVLTE
A08ARPAF APPL ACBNAME=TSOVLTF
A08ARPAG APPL ACBNAME=TSOVLTG
A08ARPAH APPL ACBNAME=TSOVLTH
A08ARPAI APPL ACBNAME=TSOVLTI
A08ARPAJ APPL ACBNAME=TSOVLTJ
```

REFERENCES

1 IBM Corporation, Advanced Communication Function for VTAM: Programming. IBM Document SC27-0449, October, 1980.

2 IBM Corporation OS/VS2 MVS JCL. IBM Document GC28-0692, May, 1979.

3 IBM Corporation, Operator's Library: OS/VS2 MVS JES2 Commands. IBM Document GC23-0007, January, 1979.

4 IBM Corporation, OS/VS2 MVS System Programming Library: JES2. IBM Document GC23-0002, January, 1979.

5 IBM Corporation, OS Assembler H Language. IBM Document GC26-3771, June, 1974.

6 IBM Corporation, OS/VS and DOS/VS Assembler Language. IBM Document GC28-6514.

7 IBM Corporation, OS PL/1 Checkout and Optimizing Compilers: Language Reference. IBM Document GC33-0009, October, 1976.

8 IBM Corporation, OS/VS2 TSO Terminal User's Guide. IBM Document GC28-0645, June, 1978.

9 IBM Corporation OS/VS2 TSO Command Language Reference. IBM Document GC28-0646, June, 1978.

10 On Line Business Systems Inc., OBS WYLBUR Reference Manual. Office of Academic Computing, UCLA, Revision, July 1979.

PART V

FAKEMSG -- DEBUGGING MSG PROGRAMS WITHOUT EXCHANGE

This section is separately available
as UCLA Document UCNSW-410.

Section 1

OVERVIEW OF FAKEMSG

FAKEMSG is a subroutine package that supports the execution of programs that use the UCLA implementation of the PL/MSG subroutine package (reference 1) in an environment where that package cannot itself be supported. In particular, FAKEMSG allows NSW server processes such as the File Package (reference 2), Foreman (reference 3), or Batch Job Package (reference 4) to be executed in the absence of the MSG Central (reference 5) package. FAKEMSG was written to allow the checkout of these processes when MSG Central was available but was unreachable due to the lack of an implementation of the UCLA inter-job communication mechanism, the EXCHANGE (reference 6).

In order to communicate with any of the processes mentioned above, it is necessary to have a calling process, usually a Works Manager (reference 7) to invoke it. Without access to the external NSW system, this need is met by the NSWDRIVE (reference 8) utility program. NSWDRIVE uses PL/MSG directly; however, the actual NSW processes all use it through the PL/PCP subroutine package (reference 9). Therefore, FAKEMSG implements just that subset of the PL/MSG interface that is used by NSWDRIVE and PL/PCP.

Section 2

STRUCTURE OF FAKEMSG

FAKEMSG operation is based upon the UCLA THREADER package (reference 10). Essentially, whenever MSG would spawn a process by submitting a batch job or by logging on a TSO session, FAKEMSG does so by creating a thread of control via the THSTART subroutine of THREADER. This requires that the routines of the processes be declared and coded as RECURSIVE, but that is already becoming a requirement of our NSW code, since we intend that it operate under THREADER for other reasons (reference 11).

FAKEMSG provides and maintains the PROCESS common datum that PL/MSG uses to identify the current MSG process. THREADER, because it was designed to be compatible with PL/MSG, also maintains that datum across thread swaps.

In keeping with THREADER's use of static storage to tie together the entities that it supervises, FAKEMSG uses static storage to anchor queues of control blocks that define MSG processes, pending events, and direct connections. Given this queue structure, implementation of most PL/MSG primitives is straight-forward.

FAKEMSG always operates with an MSG incarnation number of 1. It maintains a static instance counter that begins at 1 and increments by 1 for each process instance created. Thus the process naming conventions of FAKEMSG are compatible with those of MSG Central. Names are unique within the scope of the job containing FAKEMSG, but not across such jobs. This is compatible with the FAKEMSG design goals, since the lack of communication with MSG Central means that the universe of names can never be greater than the local job.

FAKEMSG is written in PL/I. It uses a small Assembler-language control section (FAKEMSGA) for: 1) transforming the MSGWAIT call into the THWAIT call; 2) getting and Freeing storage that will be shared by control threads, and that cannot thus be allocated directly from thread-dependent PL/I code; 3) simulating the MSGJOUR entry point through TPUT; 4) defining a table of supported generic process names and their entry points.

Section 3

SPECIFIC PL/MSG SERVICES

The sections below describe each PL/MSG entry point that has a FAKEMSG counterpart, and lists restrictions and differences in behavior. In general, any PL/MSG entry point not listed here does not reference MSG Central directly, and so may be included directly from the PL/MSG library.

Whenever a pending event is created by the entries that send or receive messages or alarms, an event-matching scan is triggered. For two events to match, these criteria must be met:

1. They must be of opposite polarity (send, receive).

2. They must be of the same type (generic, specific, alarm).

3. The receiver's generic name must match that requested by the sender (only 10 characters are checked).

4. If the type is not "generic" then the receiver's instance number must match that requested by the sender.

5. If the type is "alarm" then the receiver must be armed for alarms.

FAKEMSG does not support the case, supported by MSG central at UCLA, of a process with the "queue-generics" attribute. Generic messages to such a process are supposed to be queued for a single process instance, regardless of whether that instance is receptive to generic messages at the moment. FAKEMSG is unaware of this attribute, and it will start a new process whenever there is no "receive-generic" pending event to match a "send-generic".

3.1 MSGMP -- MATERIALIZE AN MSG PROCESS

This call is fully supported except that the "termination signal" ECB can never be posted. It creates control blocks that define a new MSG process, and sets the PROCESS shared datum accordingly. Each new process is assinged a "nickname" consisting of its generic name followed by the low-order three digits of its instance number, all truncated to 10 characters. This is sufficient to accomodate the longest generic name currently in use ("FOREMAN").

3.2 MSGSTOP -- DESTROY AN MSG PROCESS

This call is fully supported. It deletes the control blocks created by MSGMP and sets PROCESS to NULL.

3.3 MSGSETP -- DECLARE POSTING MECHANISM

This call is fully supported. It alters entries in the signal-mechanism table for the current process.

3.4 MSGAA -- ARM PROCESS FOR ALARMS

This call is fully supported.

3.5 MSGJOUR -- LOG MSG EVENTS

The user has two options here, selected by his Linkage-Editor control statements.

If he uses entry MSGJOUR of FAKEMSG, all calls to MSGJOUR will result in the passed data string being TPUT after it is concatenated behind the current process "nickname". No message filtering will occur.

If he replaces MSGJOUR with entry FAKEJOU of FAKEMSGA, the passed messages will be put out without the "nickname". In this case, calls explicitly stating a message type code between 7 and 10 (inclusive) will be suppressed. This range of message types corresponds to logging of input and output data records by the File Package.

3.6 MSGRGM -- RECEIVE GENERIC MESSAGES

This entry creates a pending event and schedules a subsequent event matching scan. It ignores its timeout parameter. Only one RGM event can be pending at a time.

3.7 MSGRSM -- RECEIVE SPECIFIC MESSAGES

This entry creates a pending event and schedules a subsequent event matching scan. It ignores its timeout parameter. The special-scheduling flags are always set to zeros. Only one RSM event can be pending at a time.

3.8 MSGRA -- RECEIVE ALARMS

This entry creates a pending event and schedules a subsequent event matching scan. It ignores its timeout parameter. Only one RA event can be pending at a time.

3.9 MSGSGM -- SEND GENERIC MESSAGES

This entry creates a pending event and schedules a subsequent event matching scan. It ignores its timeout and wait-enable parameters. Wait-enable is considered to be always set.

3.10 MSGSSM -- SEND SPECIFIC MESSAGES

This entry creates a pending event and schedules a subsequent event matching scan. It ignores its timeout and special-scheduling parameters.

3.11 MSGSA -- SEND ALARMS

This entry creates a pending event and schedules a subsequent event matching scan. It ignores its timeout parameter.

3.12 MSGRSND -- RESCIND PENDING EVENTS

This entry is a no-operation.

3.13 MSGRSNC -- RESYNCHRONIZE COMMUNICATION

This entry is a no-operation.

3.14 MSGOC -- OPEN DIRECT CONNECTIONS

This entry scans the queue of pending connections for a match on the incoming request. It ignores the bytesize and queue depth parameters. It also ignores the connection-type parameter, except that if the connection-type is "TCAM" or "VTAM" the connection is posted complete at once (for these special UCLA connections, the other partner is the user's TSO terminal, not a process). If a match is found, both processes are posted. Otherwise, a pending connection is created and left on the queue.

3.15 MSGCC -- CLOSE DIRECT CONNECTIONS

This entry is fully supported.

3.16 MSGGET -- RECEIVE CONNECTION DATA

This entry is fully supported except that it does not allow stacked calls. The caller must wait for the completion of one MSGGET before calling the entry again for the same connection. Existing MSG processes follow this practice.

3.17 MSGPUT -- SEND CONNECTION DATA

This entry is fully supported except that it does not allow stacked calls. The caller must wait for the completion of one MSGPUT before calling the entry again for the same connection. Existing MSG processes follow this practice.

3.18 MSGEOD -- PREPARE FOR CONNECTION CLOSE

This entry is a no-operation.

Section 4

CONCLUSIONS

FAKEMSG is based upon the THREADER package (reference 10). The use of PL/I and THREADER made FAKEMSG almost trivial to write.

Experience with FAKEMSG has shown us that intra-address-space MSG communication can be done simply and effectively. If we decide to move unlike MSG processes into the same address space, we will consider expanding FAKEMSG into a mechanism to effect all communication that does not need to cross address-space boundaries.

REFERENCES

1 Ludlam and Rivas, PL/MSG - An MSG Interface for PL/I. Document UCNSW-401, Office of Academic Computing, UCLA, November 15, 1980.

2 Braden and Ludlam, FP/360 - The NSW MVT File Package. Document UCNSW-204, Office of Academic Computing, UCLA, November 20, 1980.

3 Ludlam, FM/360 - The NSW MVT Foreman. Document UCNSW-205, Office of Academic Computing, UCLA, December 1, 1980.

4 Ludlam, BJP/360 - The NSW MVT Batch Job Package. Document UCNSW-207, Office of Academic Computing, UCLA, December 1, 1980.

5 Rivas, Ludlam, and Braden, An Implementation of the MSG Interprocess Communication Protocol. Document TR-12, Office of Academic Computing, UCLA, May, 1977.

6 Braden and Feigin, Programmer's Guide to the Exchange. Document TR-5, Office of Academic Computing, UCLA, March, 1972.

7 Schaffner and Sluizer, Works Manager Subsystem Specifications. Document CADD-7906-1117, Massachusetts Computer Associates, Inc., June 1, 1979.

8 Braden and Ludlam, Computing Services -- National Software Works: Final Technical Report. Document TR-27, Office of Academic Computing, UCLA, In preparation.

9 Ludlam, PL/PCP - An NSW Procedure Call Protocol Package for PL/I. Document UCNSW-402, Office of Academic Computing, UCLA, November 15, 1979.

10 Ludlam, THREADER _ A Commutator for PL/I Coroutines. Document UCNSW-409, Office of Academic Computing, UCLA, June 1, 1981.

11 Ludlam, NSW Processes Under MVS -- Basic Plan. Document UCNSW-213, Office of Academic Computing, UCLA, June 1, 1981.